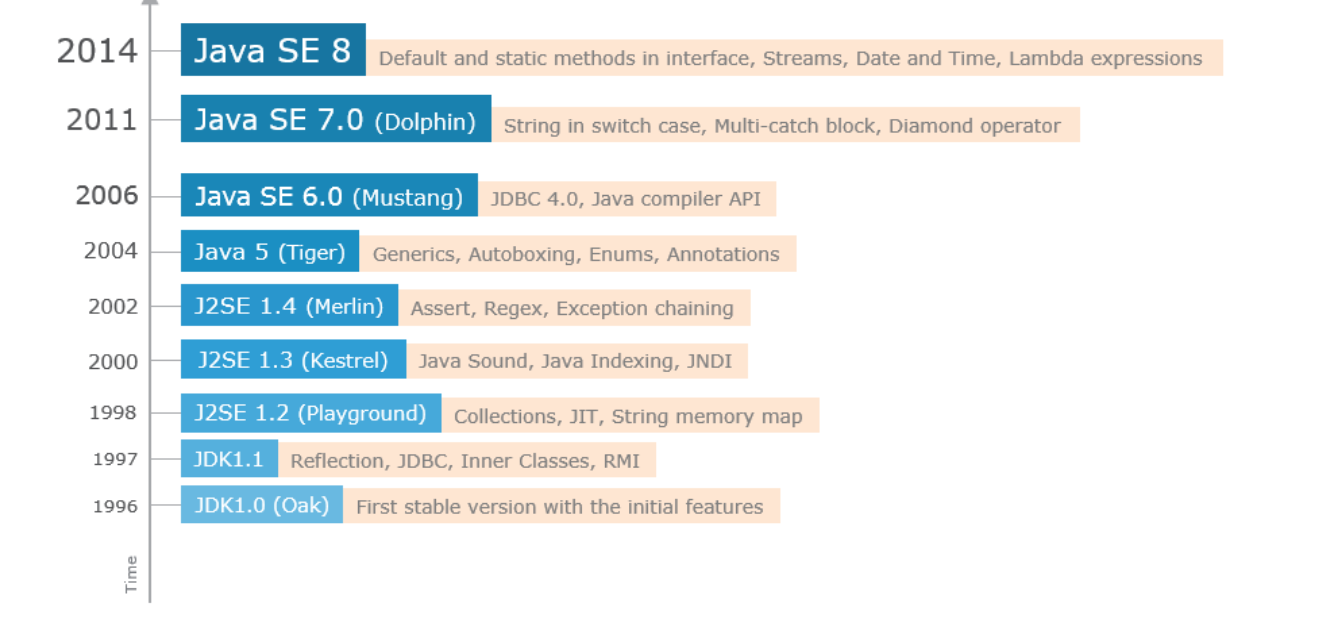
|  |
| --- |
| JAVA 8 FEATURE |

Java has been continuously evolving from its birth in 1996 as Java 1 (Oak) version until today.

In this evolution journey, Java 5 and Java 8 have been a major turning point.

This course will guide you on Java 8 features through hands on approach.



Java SE 7 has been a major milestone in the evolution history of Java, with multiple features included to shape the language further. Before we start with Java 8 features, let us glance through some features provided by Java 7:

* String literals in switch statements
* Diamond operator in Collection and Generic instance creation
* Handling multiple exceptions with a single catch block
* Try with resources block

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

|  |  |
| --- | --- |
| Before Java SE 7   1. *// gadget is a string variable* 2. if(gadget.equals("Mobile")) 3. System.out.println("Rs.5000 to 50000"); 4. else if(gadget.equals("iPad")) 5. System.out.println("Rs.10000 to 50000"); 6. else if(gadget.equals("Laptop")) 7. System.out.println("Rs.20000 to 50000"); 8. else 9. System.out.println("Not available"); | Since Java SE 7   1. *// gadget is a string variable* 2. switch(gadget) { 3. case "Mobile": System.out.println("Rs.5000 to 50000"); 4. break; 5. case "iPad": System.out.println("Rs.10000 to 50000"); 6. break; 7. case "Laptop": System.out.println("Rs.20000 to 50000"); 8. break; 9. default: System.out.println("Not available"); 10. } |

Generics are used to make collections type-safe. Prior to Java SE 7, type information had to be supplied on both the 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   1. List<String> lstNames = new ArrayList<String>(); | Since Java SE 7   1. 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 a 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   1. try { 2. *// Code that may throw an exception* 3. } 4. catch(NullPointerException ne) { 5. System.out.println(ne.getMessage()); 6. } 7. catch(StringIndexOutOfBoundsException e) { 8. System.out.println(e.getMessage()); 9. } | Since Java SE 7   1. try { 2. *// Code that may throw an exception* 3. } 4. catch(NullPointerException | StringIndexOutOfBoundsException e) { 5. System.out.println(e.getMessage()); 6. } |

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 as Automatic Resource Management (ARM).

|  |  |
| --- | --- |
| Before Java SE 7   1. Connection connection = null; 2. try { 3. connection = DriverManager.getConnection("url"); 4. *// DB operations using connection* 5. } 6. catch(SQLException e) { 7. System.out.println(e); 8. } 9. finally { 10. if(connection != null) 11. connection.close(); 12. } | Since Java SE 7   1. try(Connection connection = DriverManager.getConnection("url")) { 2. *// DB operations using connection* 3. } 4. catch(SQLException e) { 5. System.out.println(e); 6. } 7. finally { 8. *//close is not required* 9. } |

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

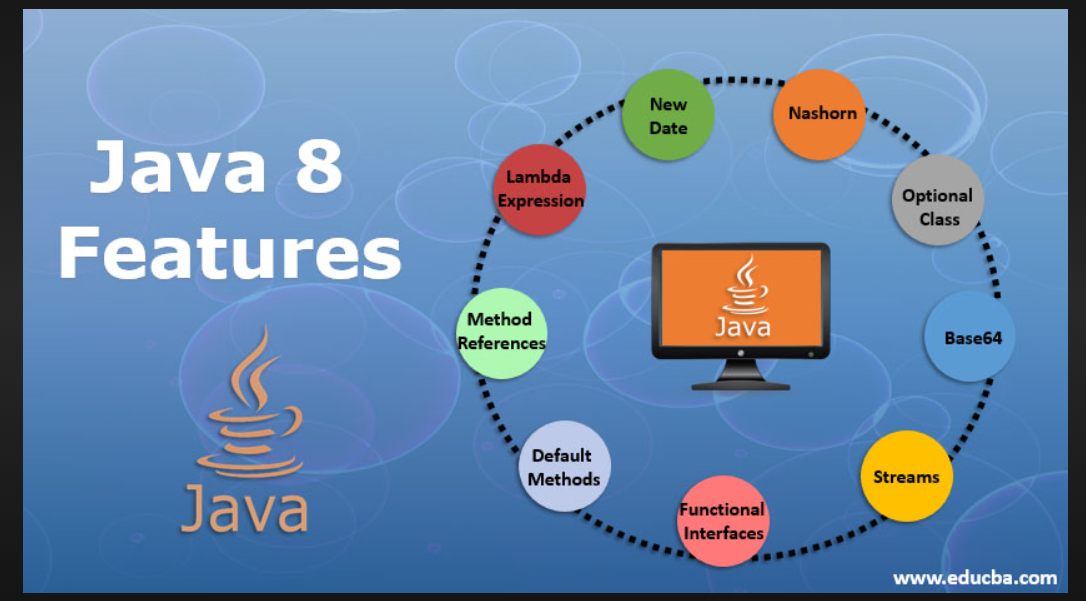
**Java Compiler**

* Named Parameters

**Java Libraries**

* Optional class
* Date/Time API
* Stream API

In this course, the features added to Java language and Java libraries will be covered in detail.



**Top Features of Java 8**

Below are the top features of Java 8 that make Java 8 more understanding and more useful:

**1. New Date/Time API**

The old Date-Time API of Java had major drawbacks. In place of it, there is a fresh Date-Time API in Java 8. We take a look at the drawbacks below:

* **Tough to Handle Timezone**: Programmers needed many lines of code to tackle timezone issues.
* **Low-Quality Design:** The earlier API had relatively few direct functions for date operations. Java 8 API offers many functions for date operations.
* **Absence of Thread Safe Property**: java.util.date lacked thread-safe property. Hence programmers had to face concurrency issues while employing data. Java 8 Date-Time API is immutable and without setter methods.

In Java 8 there is a fresh Date-Time API contained in the package java.time. Below are the 2 significant classes contained in the java. time package.

* **Zoned:** Specialized API to handle different timezones.
* **Local**: Simplified API without the complexity of handling timezones.

**2. Nashorn JavaScript**

The earlier version of Java contained the Rhino JavaScript engine. In Java 8, Rhino is replaced by a superior JavaScript engine namely Nashorn. The latter offers two to ten times superior performance when benchmarked against its predecessor. This is possible as it directly compiles the lines of code in memory followed by passing the bytecode to the Java Virtual Machine. Nashorn employs the invoke dynamics [feature of Java 7](https://www.educba.com/java-7-features/) in order to enhance performance.

**3. Optional Class**

Optional simply put is a container object which is employed to store not-null objects. One major use of the Optional object is to represent null having absent value. This important class has different utility methods to help code handle values as ‘not available’ or ‘available’ rather than to check null values. The class was added in Java 8 and is analogous to what the Optional class is in Guava.

**4. Base64**

The new version of Java contains an inbuilt encoder as well as a decoder for the purpose of Base64 encoding. Programmers can employ 3 kinds of Base64 encoding.

* **MIME**: Mapping is done of Output to the MIME format. The representation of the output is done in lines not exceeding 76 characters each. The line separator is a carriage return succeeded by a linefeed. There is no line separator at the termination of the encoded output.
* **URL**: Mapping of Output is done to a group of characters present in A-Za-z0-9+\_. The output is filename as well as the URL safe.
* **Simple**: Mapping of Output is done to a group of characters present in A-Za-z0-9+\_. No addition of any line feed in the output is done by the encoder. The decoder does accept any character differing from A-Za-z0-9+\_.

**5. Streams**

The stream is a fresh abstract layer present in this new version of Java. By employing stream, data can be processed in a declarative manner analogous to that of SQL statements.

Understanding Stream

Stream simply put is a representation of a sequence of objects emanating from a source that has support for aggregate operations. Below are certain properties of a stream.

* **Iterations are Automatic:** Explicit iterations are mandatory in Collections. Whereas in Stream iterations are done internally over the supplied source elements.
* **Pipelining:**The majority of the stream operation’s output is of the stream type. Thus, the output can be pipelined. The particular operations are termed intermediate operations. They accept input, do the necessary processing and give the output to the target.

**Some Aggregate Operations supported by Stream:**

* Match
* Find
* Reduce
* Limit
* Map
* Filter

**6. Functional Interfaces**

They display individual functionality. The new version of Java has numerous [functional interfaces](https://www.educba.com/functional-interface-in-java/) which can be employed in large measure in lambda expressions.

**7. Default Methods**

In Java 8, there is a fresh paradigm of interfaces having default method implementation. The feature is included for the purpose of backward compatibility. It is now possible to use old interfaces to harness the lambda expression capability of the new version of Java.

* **For instance:** ‘Collection’ and ‘List’ interfaces lack ‘forEach’ method declaration. If such a method is added the collection framework implementations would be broken. Now, the default method is introduced such that List/Collection contains default implementations of the forEach method. Now the respective methods do not have to be implemented by the class which is implementing these particular interfaces.

**8. Method References**

This major feature of Java 8 use is to point to relevant methods using their respective names. A “::” symbol describes method references. The latter can be employed to point to the below-mentioned kinds of methods-

* Constructors Employing the New Operator
* Instance Methods
* Static Methods

**9. Lambda Expressions**

These are claimed to be the most important as well as the most significant feature of the new version of Java. The former makes [functional programming](https://www.educba.com/functional-programming-in-java/) easy and convenient. Moreover, Lambda expressions simplify programming to a great extent.

Below is a typical lambda expression.

**parameter -> body of expression**

We take a look at the major parts of a lambda expression.

* **Return Keyword:**The value is returned by the compiler in case the body contains a single expression. Curly braces signify that the expression returns some value.
* **The parenthesis around the parameter:** If there is only a single parameter parenthesis can be omitted.
* **Type Declaration:**Parameter type declaration is not needed. From the parameter’s value, the compiler determines the necessary action.

**Miscellaneous Features of Java 8:**JDBC-ODBC Bridge has been taken off. So. has been the PermGen memory space. The ‘jjs’ command invokes the Nashorn engine while the ‘jdeps’ command analyzes the class files.

TechSol is one among the top most 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 which automates various activities that include salary calculation, deductions for the services availed, attendance and leave management etc. 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. Salary calculation is extremely different for both the categories of employees.

To have a standardized behaviour 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, 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 fee and other services provided by the company. And, the Employee class implements this interface.

 Code for the scenario:

1. public interface Remunerator {
2. double deductFoodFee();
3. }

1. public abstract class Employee implements Remunerator {
2. public abstract double calculateSalary();
3. }

|  |  |
| --- | --- |
| 1. public class FullTimeEmployee extends Employee { 2. *//field declarations* 3. public double calculateSalary() { 4. *// calculating salary - full-time employeee* 5. } 7. public double deductFoodFee() { 8. *// deducting food fee - full-time employeee* 9. } 10. } | 1. public class PartTimeEmployee extends Employee { 2. *//field declarations* 3. public double calculateSalary() { 4. *// calculating salary - part time employee* 5. } 7. public double deductFoodFee() { 8. *// deducting food fee - part time employee* 9. } 10. } |

 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 reasons but just to ensure backward compatibility.*

Here is the modified interface:

1. public interface Remunerator {
2. double deductFoodFee();
3. double HEALTH\_INSURANCE\_PERCENTAGE = 5.0;
4. double deductHealthInsurancePremium();
5. }

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.

It is common to add new functionalities to the existing APIs in the real world. But that should happen with out 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** keyword) 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 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:

1. public interface Remunerator {
2. public abstract double deductFoodFee(); *// Must be overridden by all implementing classes*
3. public final double HEALTH\_INSURANCE\_PERCENTAGE = 5.0;
4. public default double deductHealthInsurancePremium() { *// Need not be overridden*
5. *// Default implementation which can be redefined*
6. }
7. }

This is how the implementing classes will look like:

1. public class FullTimeEmployee extends Employee {
2. *//field declarations*
3. public double calculateSalary() {
4. *// calculating salary - full-time employeee*
5. }
7. public double deductFoodFee() {
8. *// deducting food fee - full-time employeee*
9. }
11. public double deductHealthInsurancePremium() { *// Default method getting overridden*
12. return (HEALTH\_INSURANCE\_PERCENTAGE \* employeeSalary) / 100;
13. }
14. }
15. public class PartTimeEmployee extends Employee { *// Default method is not getting overridden here*
16. *//field declarations*
17. public double calculateSalary() {
18. *// calculating salary - part time employee*
19. }
21. public double deductFoodFee() {
22. *// deducting food fee - part time employee*
23. }
24. }

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

* Default methods link down the variances between interfaces and abstract classes.
* They help 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 helps 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 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 |

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 8:

|  |  |
| --- | --- |
| Before Java SE 8   1. List<Employee> empList = new ArrayList<>(); 2. *// Code to add employees to empList* 3. *// Sorting empList using a comparator* 5. Collections.sort(empList, new EmployeeComparator()); | Since Java SE 8   1. List<Employee> empList = new ArrayList<>(); 2. *// Code to add employees to empList* 3. *// Sorting empList using a comparator* 5. empList.sort(new EmployeeComparator()); |

EmployeeComparator looks like below:

1. public class EmployeeComparator implements Comparator<Employee> {
2. public int compare(Employee employee1, Employee employee2) {
3. return employee1.getEmpNo().compareTo(employee1.getEmpNo());
4. }
5. }

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

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

1. interface Greeting{
2. default void hello() {
3. System.out.println(" hello from A");
4. }
5. }
6. interface GreetingExtn extends Greeting{
7. default void hello() {
8. System.out.println(" hello from B");
9. }
10. }
11. public class InheritanceProblem implements Greeting, GreetingExtn{
12. public static void main(String[] args) {
13. new InheritanceProblem().hello();
14. }
15. }

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

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

**Rule 1**: Classes always win. A method definition in the class or its super class 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.

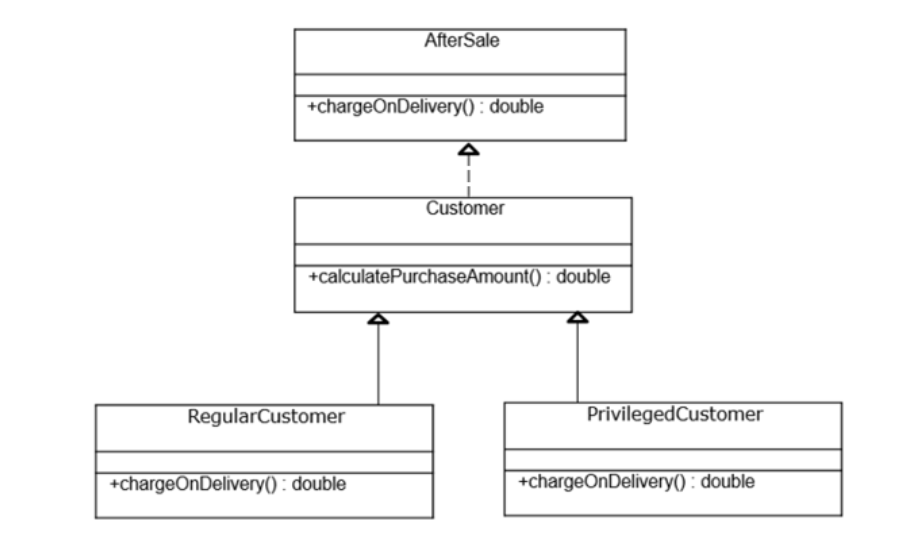
**Rule 3**: If the choice continues to be ambiguous, the class that inherits multiple interfaces should explicitly select the default method implementation to be used just by overriding it and the overridden method should have an explicit call to the desired default behavior.

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 GreetingExtn hello () method will be taken into consideration.

Dress Sharp online shopping application has two types of customers, RegularCustomer and PrivilegedCustomer (one who possesses membership card).

Interface AfterSale is designed with an abstract method called chargeOnDelivery() that needs to be overridden in both Regular and Privileged customer classes.



During Diwali, Privileged Customers are being offered an additional discount of 5%. For this, discount() method needs to be added to AfterSale interface and will be called by calculatePurchaseAmount() of Customer class. Customer class will have a list of Products which will be populated by the Client and used by calculatePurchaseAmount() to calculate the purchase amount. Arrive at a solution that will not force the RegularCustomer to implement the discount() method and will apply discount for the previleged customer alone.

Scenario-2:

 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.

1. public class CommonDeductions { *// Class with methods for calculating deductions common to all employee categories*
2. final double PENSION\_PERCENTAGE = 5.0;
4. public static double deductForPension(double employeeSalary) {
5. return (employeeSalary \* PENSION\_PERCENTAGE / 100);
6. }
7. }

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.

 Static Methods:

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:

1. public interface Remunerator {
2. public abstract double deductFoodFee(); *// Must be overridden by all implementing classes*
3. final double HEALTH\_INSURANCE\_PERCENTAGE = 5.0;
4. final double PENSION\_PERCENTAGE = 5.0;
6. public default double deductHealthInsurancePremium() { *// Need not be overridden*
7. *// Default implementation which can be redefined*
8. }
10. public static double deductForPension(double employeeSalary) { *// Static method of the interface*
11. return (employeeSalary \* PENSION\_PERCENTAGE / 100);
12. }
13. }

The static method can be used as follows:

1. public class PartTimeEmployee extends Employee {
2. *//field declarations*
3. public double calculateSalary() {
4. *// calculating salary - part time employee*
6. this.employeeSalary -= Remunerator.deductForPension(this.employeeSalary);
8. *// Further operations*
9. }
11. public double deductFoodFee() {
12. *// Food fee deduction from salary*
13. }
14. }

Here are some observations about the static methods of interfaces:

* Static methods in interfaces help providing 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:

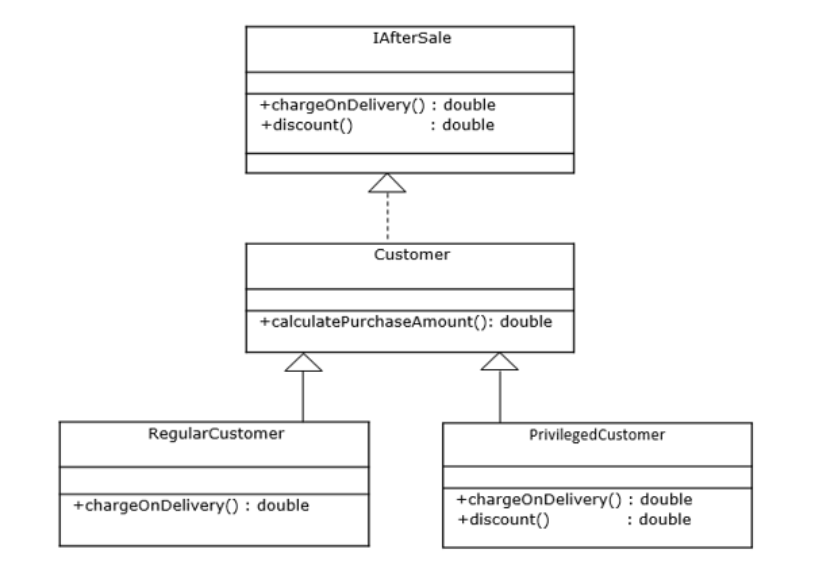
1. public class SortStaticDemo {
2. public static void main(String[] args) {
3. List<String> countrylst = Arrays.asList("India", "America", "Japan", "Brazil");
4. countrylst.sort(Comparator.naturalOrder()); *// will sort in String natural sorting order*
5. for (String countryName : countrylst) {
6. System.out.println(countryName);
7. }
8. }
9. }

**Output:**

1. America
2. Brazil
3. India
4. Japan

## Problem Statement:

Dress Sharp online application wants to issue gifts to its customers.



Interface IAfterSale needs to be added with a method called issueGift() that decides the gift to be issued to the customers based on their purchase amount. The business logic of this method is common for both the type of customers. The issueGift() method will be called by calculatePurchaseAmount() internally. Customer class will have a list of Products which will be populated by the Client and used by calculatePurchaseAmount() to calculate the purchase amount.

Arrive at a solution that will satisfy this requirement.

Scenario-1

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 the java.util package:

1. public class CountryNameComparator implements Comparator<Employee> {
2. *//overriding compare method --- sorting happens ---- country based*
3. public int compare(Employee employee1, Employee employee2) {
4. return employee1.getCountry().compareTo(employee2.getCountry());
5. }
6. }

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

1. public void sortEmployeesByCountry(List<Employee> empList) {
2. *//passing sorting logic as the parameter*
3. empList.sort(new CountryNameComparator());
4. }

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:

1. public void sortEmployeesByCountry(List<Employee> empList) {
2. *//passing inner class --- anonymous*
3. empList.sort(new Comparator<Employee>() {
4. public int compare(Employee employee1, Employee employee2) {
5. return employee1.getCountry().compareTo(employee2.getCountry());
6. }
7. });
8. }

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 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:

1. public void sortEmployeesByCountry(List<Employee> empList) {
2. empList.sort((Employee employee1, Employee employee2) -> employee1.getCountry().compareTo(employee2.getCountry()));
3. }

This code shall be written as:

1. Comparator<Employee> comparator = (Employee employee1, Employee employee2) -> employee1.getCountry().compareTo(employee2.getCountry());
2. 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**: 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:
   1. () -> { System.out.println("No argument"); }
   2. (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.

* 1. argument -> { return argument\*argument; }

1. The **arrow** (->) token
2. The **body**: Single statement or a block.  
   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.
   1. (e1, e2) -> e1.getCountry().compareTo(e2.getCountry())

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

* 1. (e1, e2) -> {
  2. int value = e1.getCountry().compareTo(e2.getCountry());
  3. return value;
  4. }

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

## Problem Statement:

Change the following anonymous Runnable class implementation to Lambda Expression:

1. public class DemoThreadMine {
2. public static void main(String pars[]) {
3. Thread threadInstance = new Thread(new Runnable() {
4. *//run --- implementation*
5. public void run() {
6. System.out.println(" Its me from thread");
7. }
8. });
9. threadInstance.start();
10. }
11. }

As you have noticed, lambda expressions provide interface implementations.  
Therefore, the target type of any lambda expression is of **Functional Interfaces**.  
Functional interfaces strictly have abstract methods of count one. However, they are allowed to have any number of static or default methods. In addition, they can override some methods from java.lang.Object.

Hence, lambda expressions can appear only in places where functional interfaces are used.

**@FunctionalInterface** annotation helps designing an interface as functional interface. Doing so, will make the compiler raise an error if the count of abstract methods in the interface exceeds one.

1. @FunctionalInterface
2. interface Shape {
3. void draw(); *// The only abstract method*
4. }

While working with lambda expressions, the compiler is responsible for inferring its type. For example:

1. Collections.sort(empList , (employee1, employee2) -> employee1.getCountry().compareTo(employee2.getCountry()));

The compiler will be able to refer the above lambda expression based on the following:

* Based on the current context, compiler infers, second parameter should be of type java.util.Comparator Interface. This inference has happened based on the definition of Collections.sort() method.
* java.util.Comparator has exactly one abstract method compare() and can be used as the second parameter of sort().
* The argument list of lambda expression (employee1, employee2) matches the compare(Object objt1, Object objt2) method present in the Comparator interface.
* The return type of lambda's body is int, that exactly matches the compare() method's return type.
* The body of the lambda here, throws no checked exception, and hence matches compare() method's complete signature.

Some of the functional interfaces available in Java are as follows:

|  |  |
| --- | --- |
| **Functional Interface** | **Abstract Method** |
| java.awt.event.ActionListener | void actionPerformed(ActionEvent event) |
| java.beans.PropertyChangeListener | void propertyChange(PropertyChangeEvent evt) |
| java.io.FileFilter | boolean accept(File filePathname) |
| java.lang.Runnable | void run() |
| java.security.PrivilegedAction | T run() |
| java.util.concurrent.Callable | V call() |
| java.util.Comparator | int compare(T objectt1, T objectt2) |

## ProblemStatement:

Define a functional interface called **StringFormatter**that should contain the abstract method,

1. String format(String string1, String string2);

Create a main class that contains lambda expressions to implement the StringFormatter interface by defining the format() method in the following three different ways for the inputs: s1 = “Lambda”, s2 = “Expression”

1. Returns "Lambda Expression"
2. Returns "Lambda – Expression"
3. Returns "LAMBDA EXPRESSION"

Let us put our logic to compare employees by their countries as a new method in our Employee class:

1. class Employee {
2. private Integer empId;
3. private String empName;
4. private String country;
6. *// Other fields, constructor, mutator and accessor methods*
8. public static int compareByCountry(Employee employee1, Employee employee2) {
9. return employee1.getCountry().compareTo(employee2.getCountry());
10. }
11. }

The Lambda to use this would look like:

1. empList.sort((employee1, employee2) -> Employee.compareByCountry(employee1, employee2));

In such cases where a lambda expression simply calls an existing method, it can be written as:

1. empList.sort(Employee::compareByCountry);

This is the **method reference** syntax for lambda expressions, which makes the expression compact and easy to read.  
It can be used whenever methods already exist with a name.

Method references can be applied in the following ways:

|  |  |  |
| --- | --- | --- |
| **Method Type** | **Lambda Expression** | **Method Reference** |
| Static method | (String s) -> Integer.parseInt(s) | Integer::parseInt |
| Instance method of particular type | (String s) -> s.length() | String::length |
| Instance method of particular type | () -> emp.getEmpId()  where emp is an instance of Employee | emp::getEmpId |
| Constructor | (String s)->new String(s) | String::new |

Scenario-1

TechSol HR is coming up with a lucky draw scheme for its employees. For this, it needs to divide the employees into two groups - one with even employee numbers, and the other with odd.

Here we have two methods to display the two groups:

1. public class TechSolLuckyDraw {
2. public static void main(String[] args) {
3. *// lstEmpIds is the populated list of employee ids*
4. printEvenIds(lstEmpIds);
5. printOddIds(lstEmpIds);
6. }
8. public static void printEvenIds(List<Integer> lstEmpIds) {
9. for(int id : lstEmpIds) {
10. if(id % 2 == 0) System.out.println("Even Employee Id: " + id);
11. }
12. }
14. public static void printOddIds(List<Integer> lstEmpIds) {
15. for(int id : lstEmpIds) {
16. if(id % 2 != 0) System.out.println("Odd Employee Id: " + id);
17. }
18. }
19. }

Now, TechSol wants to give stock options to the employees who are associated with the company for more than 20 years. Employee numbers 20 years ago were less than 1500.

The following method shall be included to display the desired list of employees:

1. public class TechSolLuckyDraw {
2. public static void main(String[] args) {
3. *// lstEmpIds is the populated list of employee ids*
4. printEvenIds(lstEmpIds);
5. printOddIds(lstEmpIds);
6. printIdsForStock(lstEmpIds);
7. }
9. public static void printEvenIds(List<Integer> lstEmpIds) {
10. for(int id : lstEmpIds) {
11. if(id % 2 == 0) System.out.println("Even Employee Id: " + id);
12. }
13. }
15. public static void printOddIds(List<Integer> lstEmpIds) {
16. for(int id : lstEmpIds) {
17. if(id % 2 != 0) System.out.println("Odd Employee Id: " + id);
18. }
19. }
21. public static void printIdsForStock(List<Integer> lstEmpIds) {
22. for(int id : lstEmpIds) {
23. if(id < 1500) System.out.println("Employee Id less than 1500: " + id);
24. }
25. }
26. }

As we see, the code is modified for each fresh/changing requirement.

As the functionalities in all the methods seem to be similarly targeted, i.e. filtering employees, would it not be better to have a single method perform the functionalities that could be passed to it?

This is possible using lambda expressions, and contributes as its most important advantage.  
Our case needs a functional interface that tests conditions – Predicate.

Let us see how this affects our code:

1. import java.util.\*;
2. import java.util.function.Predicate;
4. public class TechSolLuckyDraw {
5. *//main method*
6. public static void main(String[] args) {
7. *// lstEmpIds is the populated list of employee ids*
8. System.out.println("Printing Even Employee Ids");
9. evaluate(lstEmpIds, (n) -> n % 2 == 0);
10. System.out.println("Printing Odd Employee Ids");
11. evaluate(lstEmpIds, (n) -> n % 2 == 1);
12. System.out.println("Printing Employee Ids less than 1500");
13. evaluate(lstEmpIds, (n) -> n < 1500);
14. }
16. public static void evaluate(List<Integer> listEmpId, Predicate<Integer> predicateArg) {
17. for(int empId : listEmpId) {
18. if(predicateArg.test(empId))
19. System.out.println(empId);
20. }
21. }
22. }

As you can see, our generic evaluate() method performs the functionality/behavior passed to it.  
Let us discuss how does it works.

In our solution, we have used java.util.function.Predicate which is interestingly a functional interface that was introduced in Java 8.  
It has an abstract method test() returning a boolean value as shown.

1. @FunctionalInterface
2. public interface Predicate<T> {
3. *// some default and static methods*
4. boolean test(T testPar);
6. }

In our example for the even number scenario, we pass **n % 2 == 0** to the evaluate() method, and this becomes the implementation for the test() method.  
So for each id in the list, the test() method will return true only when id % 2 == 0.

Similarly we can pass different implementations for the test() method using lambda expressions.

There are many other useful functional interfaces defined in the java.util.function package.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Functional Interface** | **Abstract Method** | **Description** |
| Function | Function<T, R> | R apply(T t) | Function that accepts single argument and produces result |
| Predicate | Predicate<T> | boolean test(T t) | Boolean-valued function that takes single argument |
| Consumer | Consumer<T> | void accept(T t) | Function that accepts single argument but returns no result |
| Supplier | Supplier<T> | T get() | Function that denotes a supplier of results |

We have previously covered Predicate interface. Now let us cover others as well.

Every time TechSol decides to provide salary increment to its employees, a report with old and new salaries needs to be generated.

In this scenario, we need to accept the employee object as input and provide the incremented salaries as the result.  
The Function interface fits in here, which is generally used when input and result are of different types.

1. import java.util.List;
2. import java.util.function.Function;
4. public class EmployeeService {
5. *//main method*
6. public static void main(String pars[]) {
7. List<Employee> lstEmp = Employee.getEmpList();
8. for (Employee e : lstEmp)  {
9. double sal = computeSalary(e, e1 -> e1.getSal() \* 1.10); *// 10% salary increment*
10. System.out.println(e.getName() + ": old salary: " + e.getSal() + ", new salary: " + sal);
11. }
12. }
14. public static double computeSalary(Employee e, Function<Employee, Double> fn) {
15. return fn.apply(e);
16. }
17. }

Here we define computeSalary method with Employee e and Function fn which takes Employee as input and Double as return value. We pass **e1.getSal() \* 1.01** to the computeSalary() method in order to increment the salary by 10%, and this becomes the implementation for apply() method. So for each employee passed the salary increment is applied using fn.apply().

Since we are using lambda to pass the logic for increment as a parameter, we can easily change the logic to provide a different value of increment, 8% for example.

TechSol needs to generate a report with the details of employees.  
For the HR team, salary component should be included, whereas for managers, the report should not have the salary component.

In this scenario, we need a functional interface, which will accept a parameter without returning any value. The Consumer interface can be used which has an abstract accept() method:

1. import java.util.List;
2. import java.util.function.Consumer;
4. public class EmployeeService {
5. public static void main(String[] args) {
6. List<Employee> lstEmp = Employee.getEmpList();
8. System.out.println("\*\*\*\*\* Generating HR Report \*\*\*\*\*");
9. generateReport(lstEmp, e -> System.out.println(e.getId() + " : " + e.getName() + " : " + e.getRole() + " : " + e.getSal()));
11. System.out.println("\n\*\*\*\*\* Generating Manager Report \*\*\*\*\*");
12. generateReport(lstEmp, e -> System.out.println(e.getId() + " : " + e.getName() + " : " + e.getRole()));
13. }
15. public static void generateReport(List<Employee> lstEmp, Consumer<Employee> consumer) {
16. for (Employee e : lstEmp) {
17. consumer.accept(e);
18. }
19. }
20. }

Here we pass **System.out.println(…)** for the generateReport() method in order to generate the report, and this becomes the implementation for the accept() method. So for the list of employee passed report is generated using consumer.accept().

Since we are using lambda to pass the logic for report generation, we can easily change the logic to generate report for managers by providing a different implementation as seen in the second call of the generateReport() method.

The finance team at TechSol needs to generate the salary slip for all the employees, i.e. both full time and part time employees.

As we have seen earlier, the FulltimeEmployee and PartTimeEmployee classes extend from Employee class. We have added generateSalarySlip() method in these classes to generate the salary slip. For the given scenario the Supplier interface can be used which has an abstract get() method:

1. import java.util.List;
2. import java.util.function.Supplier;
4. public class SalaryGenerator {
5. public static void main(String[] args) {
6. List<? extends Employee> lstEmp = Employee.getEmpList();
7. for (Employee e : lstEmp) {
8. generate(() -> e);
9. }
10. }
12. public static void generate(Supplier<? extends Employee> supplier) {
13. supplier.get().generateSalarySlip();
14. }
15. }

Here we pass **e** (instance variable of employee) to the generate() method in order to generate the salary slip, and this becomes the implementation for the get() method. So for each employee, irrespective of full time or part time, supplier.get() returns an instance of Employee on which the generateSalarySlip() method is called.

## Problem Statement:

Write a Calculator class using lambda expressions to add, subtract and multiply two int values.

1. import java.util.function.BiFunction;
3. public class CalculatorQuiz {
4. *//main*
5. public static void main (String pars[]) {
6. *// call evaluate for adding two int values*
7. *// call evaluate for subtracting two int values*
8. *// call evaluate for multiplying two int values*
9. }
11. public static Integer evaluate(Integer t, Integer u, BiFunction<Integer, Integer, Integer> fn) {
12. return fn.apply(t, u);
13. }
14. }

Remember traversing collections using iterators? It requires the developer to take care of:

* **How** - creating the iterator, looping through it, and checking for more elements
* **What** - the business functionality

Similar steps need to be followed while using the for-each loop:

1. for (Employee emp : empList) {
2. System.out.println(emp.getCountry());
3. }

This is a perfect example for external iteration.  
Don't you think it would have been easier if the developer could concentrate only on the **What** part?

Java provides internal iteration using Lambda expressions to solve this problem.  
forEach() method of Collection interface will fit this scenario.

1. empList.forEach(emp -> System.out.println(emp.getCountry()));

This allows the developer to focus on what needs to be done while doing away with other boilerplate code.  
The above is possible due to the addition of a default method to Iterable Interface in Java 8. It accepts a Consumer functional interface:

1. default void forEach(Consumer<? super T> action)

Internal iteration also allows parallel processing instead of a sequential one and shall be discussed later in this course.

## Problem Statement:

Modify the given class to use internal iteration to print all the country names in ascending order.

1. import java.util.\*;
3. public class CountryCaseChange {
4. public static void main(String[] args) {
5. List<String> listCountries = Arrays.asList("France", "Germany", "Italy", "Egypt");
6. countrylst.sort((String stringArg1, String stringArg2) -> stringArg1.compareTo(stringArg2));
7. for (String countryName : listCountries) {
8. System.out.println(countryName);
9. }
10. }
11. }

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:

1. import java.util.\*;
3. public class FilterEmployee {
4. public static void main(String[] args) {
5. List<Employee> lstEmp = Employee.getEmpList();
6. List<Employee> listNewEmp = new ArrayList<>();
7. for (Employee emp : lstEmp) {
8. if (emp.getYearsInOrg() < 1) { *// Filtering*
9. listNewEmp.add(emp);
10. }
11. }
13. Collections.sort(listNewEmp, new Comparator<Employee>() { *//logic for sorting*
14. public int compare(Employee employee1, Employee employee2) {
15. return employee1.getId() - employee2.getId();
16. }
17. });
19. for (Employee newEmp : listNewEmp) {
20. System.out.println(newEmp.getId() + ":" + newEmp.getName()); *// content display*
21. }
22. }
23. }

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

1. import java.util.\*;
3. public class FilterEmployee {
4. public static void main(String[] args) {
5. List<Employee> lstEmp = Employee.getEmpList();
6. List<Employee> lstNewEmp = new ArrayList<>();
8. lstEmp.forEach((e) -> { if(e.getYearsInOrg() < 1) *// Filtering*
9. lstNewEmp.add(e);
10. });
11. lstNewEmp.sort((employee1,employee2) -> employee1.getId() - employee2.getId()); *// Sorting*
12. lstNewEmp.forEach((e) -> System.out.println(e.getId() + ":" + e.getName())); *// Displaying*
13. }
14. }

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:

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

From arrays, streams can be built as follows:

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

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

1. String fileName = "C://Employees.txt";
2. *// Reading file into stream inside try-with-resources*
3. try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
4. stream.forEach(System.out::println);
5. } catch (IOException excIO) {
6. e.printStackTrace();
7. }

Streams can also be built simply from values as below:

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

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

1. Stream<Employee> empStream = lstEmp.stream();
2. 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:

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

1. Stream<Employee> empStream = lstEmp.stream();
2. Stream<Employee> newEmpStream = empStream.filter(emp -> emp.getYearsInOrg() < 1);
3. 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:

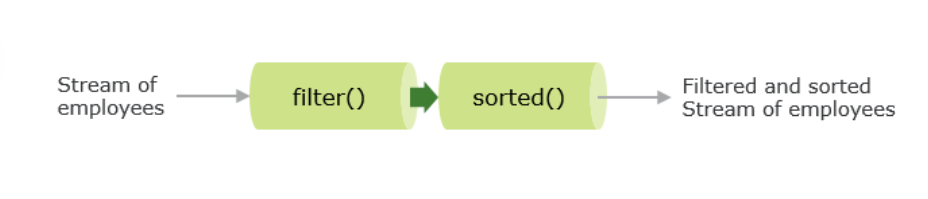
1. Stream<T> sorted(Comparator<? super T> comparator);

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

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:

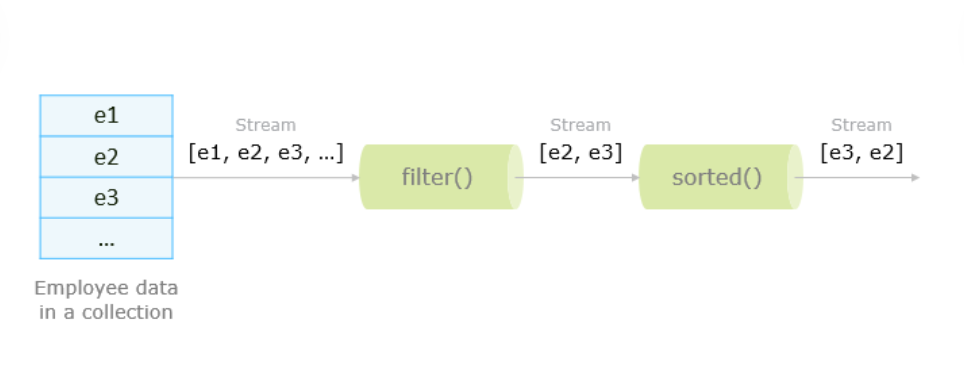
1. Stream<Employee> sortedEmpStream = lstEmp.stream()
2. .filter(emp -> emp.getYearsInOrg() < 1)
3. .sorted((e1, e2) -> e1.getId() - e2.getId());

The solution to our scenario using streams would now be:

1. import java.util.\*;
3. public class FilterEmployee {
4. public static void main(String[] args) {
5. List<Employee> lstEmp = Employee.getEmpList();
7. lstEmp.stream().filter(emp -> emp.getYearsInOrg() < 1)
8. .sorted((e1, e2) -> e1.getId() - e2.getId())
9. .forEach((e) -> System.out.println(e.getId() + ":" + e.getName()));
10. }
11. }

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.

## Problem Statement:

Consider the Employee class as shown:

1. public class Employee {
2. public int id;
3. public String name;
4. private int sal;
5. private double yearsInOrg;
6. private String role;
7. private String gender;
9. public Employee(String name, int id, int sal, double years, String role, String gender) {
10. this.id = id;
11. this.name = name;
12. this.sal = sal;
13. this.yearsInOrg = years;
14. this.role = role;
15. this.gender = gender;
16. }
18. *// Getters and setters*
19. }

For a set of employees, print the details for following scenarios:

a. Employees whose role is TA

b. Count of female employees

map() Method:

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

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

1. import java.util.\*;
2. import java.util.stream.Collectors;
3. public class FilterEmployee {
4. public static void main(String[] args) {
5. List<Employee> lstEmp = Employee.getEmpList();
6. lstEmp.forEach((e) -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
7. List<Employee> lstNewEmp = lstEmp.stream().filter(emp -> emp.getYearsInOrg() < 1)
8. .map(e -> { e.setSal(e.getSal() + 5000); return e; })
9. .collect(Collectors.toList());
10. lstNewEmp.forEach((e) -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
11. }
12. }

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

1. <R> Stream<R> map(Function<? super T, ? extends R> mapper);

The function is applied on each element and mapped into 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:

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

It accepts a Collector type as 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.

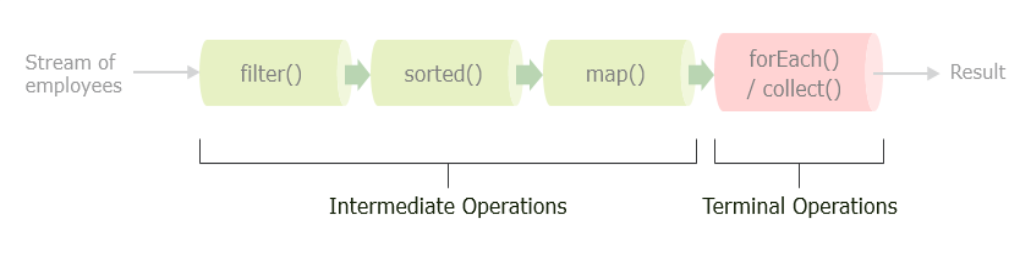
## Problem Statement:

Given a list of numbers, return a list of the cube of each number.

For example, given **[1, 2, 3, 4, 5]** the program should return **[1, 8, 27, 64, 125]**.

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 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:  lstEmp.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):  lstEmp.stream().distinct(); |

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

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

This can be solved as below:

1. public class FilterEmployee {
2. public static void main(String[] args) {
3. List <Employee> lstEmp = Employee.getEmpList();
5. List<Employee> lstNewEmp = lstEmp.stream().filter(emp -> emp.getYearsInOrg() < 1)
6. .map(e -> { e.setSal(e.getSal() + 5000); return e; }) *// Incrementing salary*
7. .collect(Collectors.toList());
9. System.out.println("Employees less than a year old with increment:");
10. lstNewEmp.forEach(e -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
12. Optional<Integer> max = lstNewEmp.stream().map(e->e.getSal())
13. .reduce(Integer::max); *// Finding the maximum salalry*
14. List<Employee> lstMaxEmp = lstNewEmp.stream()
15. .filter(e -> e.getSal() == max.get()) *// Finding employees with the maximum salary*
16. .collect(Collectors.toList());
18. System.out.println("\nEmployees having maximum salary after increment:");
19. lstMaxEmp.forEach(e -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
20. }
21. }

1. Employees less than a year old with increment:
2. 34578:Cathy Ivy:35000
3. 41234:Damodar Charan:61000
4. 22347:Netaa Singh:100000
6. Employees having maximum salary after increment:
7. 22347:Netaa Singh:100000

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

1. 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.

## Problem Statement:

Calculate the sum of the first 50 whole numbers using iterate() and reduce() methods of Stream.

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:

1. public class FilterEmployee {
2. public static void main(String[] args) {
3. List<Employee> lstEmp = Employee.getEmpList();
4. lstEmp.parallelStream().filter(emp -> emp.getYearsInOrg() < 1)
5. .sorted((e1, e2) -> e1.getId() - e2.getId())
6. .forEach((e) -> System.out.println(e.getId() + ":" + e.getName()));
7. }
8. }

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 which are dependent on the ordering of the elements like limit(), findFirst(), etc. are quite expensive in parallel stream.