**Functional Interface :-** An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

*package* java8.functionalInterface;  
@FunctionalInterface  
*public interface ParentFunctionalInterface* {  
 *public void* print();  
}

*package* java8.functionalInterface;  
@FunctionalInterface  
*public interface ChildFunctionalInterface extends ParentFunctionalInterface* {  
  
 *//If we extend funtional interface to another functional interface then child interface can't contain any abstract method.  
 public void* childPrint();//compilation error  
  
 *default void* defaultChildMethod() {  
 System.*out*.println("default method");  
 }  
  
 *static void* staticChildMethod() {  
 System.*out*.println("static method");  
 }  
  
 *public class* FunctionalInterfaceInheritance *implements ChildFunctionalInterface* {  
  
 @Override  
 *public void* print() {  
 System.*out*.println("Print method of ParentFunctionalInterface.");  
 }  
 }  
}

**Output :- Compilation error -** Class 'FunctionalInterfaceInheritance' must either be declared abstract or implement abstract method 'childPrint()' in 'ChildFunctionalInterface.

**Note :-** If we implement childPrint() method inside FunctionalInterfaceInheritance then compile error message would be “Multiple non-overriding abstract methods found in interface java8.functionalInterface.ChildFunctionalInterface”.

**Consumer Functional Interface :-** It is a functional interface defined in java.util.function package. It contains an abstract accept() and a default andThen() method. It can be used as the assignment target for a lambda expression or method reference.

The Consumer Interface accepts a single argument and does not return any result.

*package* java8.Consumer;  
*import* java.util.function.*Consumer*;  
  
*public class* FirstConsumerExample {  
  
 *public static void* main(String[] args) {  
  
 *Consumer*<String> consumerString = string -> System.*out*.println(string);  
 consumerString.accept("Mona");  
  
 System.*out*.println("================================================================");  
  
 *Consumer*<String> printUpperCase = (upperCaseString) -> System.*out*.println(upperCaseString.toUpperCase());  
 printUpperCase.accept("Diksha");  
  
 System.*out*.println("================================================================");  
  
 *Consumer*<String> reverseConsumerString = reverseString -> {  
 StringBuilder sb = *new* StringBuilder(reverseString);  
 System.*out*.println(sb.reverse());  
 };  
 consumerString.andThen(printUpperCase).andThen(reverseConsumerString).accept("Shikha");  
 }  
 }

**Output :-**

Mona

================================================================

DIKSHA

================================================================

Shikha

SHIKHA

ahkihS

**Supplier Functional Interface :-** Supplier is functional interface which does not take any argument and produces result of ***type T***. It has a functional method called ***T get()*** As Supplier is functional interface, so it can be used as assignment target lambda expressions.

*package* java8.supplier;  
*import* java.util.Random;  
*import* java.util.function.*Supplier*;  
  
*public class* FirstSupplierExample {  
  
 *public static void* main(String[] args) {  
 *Supplier*<String> stringSupplier = () -> {  
 *return* "This is first example of supplier interface.";  
 };  
  
 System.*out*.println(stringSupplier.get());  
  
 *Supplier*<Integer> intSupplier = () -> {  
 *return new* Random().nextInt(100);  
 };  
  
 System.*out*.println(intSupplier.get());  
 }  
}

**Output :-**

This is first example of supplier interface.

7

**Note :-** 7 is the random number. It can be anything.

**Function Functional Interface :-** *java.util.function.Function* is a functional interface whose functional method (single abstract method) is *R apply(T t)*.

It takes an argument (object of type T) and returns an object (object of type R). The argument and output can be a different type.

*package* java8.functionInterface;  
*import* java.util.function.*Function*;  
  
*public class* FirstFuntionInterface {  
  
 *public static void* main(String[] args) {  
 FirstFuntionInterface firstFuntionInterface = *new* FirstFuntionInterface();  
 *Function*<Integer, Boolean> function = firstFuntionInterface :: checkAgeVaidity;  
 System.*out*.println(function.apply(23));  
 }  
  
 *public* Boolean checkAgeVaidity(Integer age) {  
 *return* age > 18 ? Boolean.*TRUE* : Boolean.*FALSE*;  
 }  
}

**Output :-**

True

*package* java8.functionInterface;  
*import* java.util.function.*Function*;  
  
*public class* SecondFunctionInterface {  
 *//In this class we will test 'andThen' method and 'compose' method of Function interface.  
 public static void* main(String[] args) {  
  
 *Function*<Integer,Integer> add = x -> x+3;  
  
 *Function*<Integer, Integer> multiply = x -> x\*2;  
  
 *//Below example of 'andthen' method. 'add' Funtion interface call 'andThen' method and passing 'multiply'  
 // Funtion interface as parameter. At the end call 'apply' method and passing 2 as input integer.  
 // First 'add' Function interface 'apply' method call and then 'andThen' Function interface method call* Integer resultandThen = add.andThen(multiply).apply(2);  
 System.*out*.println("andThen result ::: "+resultandThen);  
   
 *//Below example of 'compose' method. 'add' Funtion interface call 'compose' method and passing 'multiply'  
 // Funtion interface as parameter. At the end call 'apply' method and passing 2 as input integer.  
 // If 'add' Function interface call 'compose' method and passing another Function interface that means  
 // another Function interface call first and then 'add'. In that case 'mutiply' is another Function interface.  
 // So 'multiply' Function interface call 'apply' method then 'add' Function interface call 'apply' method.* Integer resultcompose = add.compose(multiply).apply(2);  
 System.*out*.println("compose result ::: "+resultcompose);  
   
 *//Note : 'compose' method just opposite work of 'andThen' method. This is called function chaining.  
 }*  
}

**Output :-**

andThen result ::: 10

compose result ::: 7

**Predicate Functional Interface :-** It is a functional interface which represents a predicate (boolean-valued function) of one argument. It is defined in the java.util.function package and contains test() a functional method. Usually, it used to apply in a filter for a collection of objects.

**BiPredicate Functional Interface :-** In Java 8, BiPredicate is a functional interface, which accepts two arguments and returns a boolean, basically this BiPredicate is same with the Predicate, instead, it takes 2 arguments for the test.

*package* java8.predicate;  
  
*import* java.util.Arrays;  
*import* java.util.*List*;  
*import* java.util.function.*BiPredicate*;  
*import* java.util.function.*Predicate*;  
*import* java.util.stream.Collectors;  
  
*class* EmployeePredicate {  
  
 *private* Integer id;  
 *private* Integer age;  
 *private* String gender;  
 *private* String firstName;  
 *private* String lastName;  
  
 *public* EmployeePredicate(Integer id, Integer age, String gender, String firstName, String lastName) {  
 *this*.id = id;  
 *this*.age = age;  
 *this*.gender = gender;  
 *this*.firstName = firstName;  
 *this*.lastName = lastName;  
 }  
  
 *public* Integer get() {  
 *return* id;  
 }  
  
 *public void* setId(Integer id) {  
 *this*.id = id;  
 }  
  
 *public* Integer getAge() {  
 *return* age;  
 }  
  
 *public void* setAge(Integer age) {  
 *this*.age = age;  
 }  
  
 *public* String getGender() {  
 *return* gender;  
 }  
  
 *public void* setGender(String gender) {  
 *this*.gender = gender;  
 }  
  
 *public* String getFirstName() {  
 *return* firstName;  
 }  
  
 *public void* setFirstName(String firstName) {  
 *this*.firstName = firstName;  
 }  
  
 *public* String getLastName() {  
 *return* lastName;  
 }  
  
 *public void* setLastName(String lastName) {  
 *this*.lastName = lastName;  
 }  
  
 @Override  
 *public* String toString() {  
 *return* "EmployeePredicate{" +  
 "id=" + id +  
 ", age=" + age +  
 ", gender='" + gender + '\'' +  
 ", firstName='" + firstName + '\'' +  
 ", lastName='" + lastName + '\'' +  
 '}';  
 }  
}  
  
*public class* PredicateEmployeeExample {  
  
 *public static void* main(String[] args) {  
 EmployeePredicate e1 = *new* EmployeePredicate(1,23,"M","Ricky","Luther");  
 EmployeePredicate e2 = *new* EmployeePredicate(2,22,"F","Rosy","Taylor");  
 EmployeePredicate e3 = *new* EmployeePredicate(3,33,"M","Jayden","West");  
 EmployeePredicate e4 = *new* EmployeePredicate(4,21,"M","Shane","Lawmen");  
 EmployeePredicate e5 = *new* EmployeePredicate(5,19,"F","Cristine","Josie");  
 EmployeePredicate e6 = *new* EmployeePredicate(6,25,"M","David","Feezor");  
 EmployeePredicate e7 = *new* EmployeePredicate(7,38,"F","Scarlet","Ellie");  
 EmployeePredicate e8 = *new* EmployeePredicate(8,41,"M","Russel","Gussin");  
 EmployeePredicate e9 = *new* EmployeePredicate(9,35,"F","Rakul","Singh");  
 EmployeePredicate e10 = *new* EmployeePredicate(10,45,"M","Aiden","Taylor");  
  
 *List*<EmployeePredicate> employees = Arrays.*asList*(e1,e2,e3,e4,e5,e6,e7,e8,e9,e10);  
 System.*out*.println("All employees ::: "+employees);  
 System.*out*.println("=========================================================================");  
  
 *//We have to filter young employee which age > 21 and age < 40  
 //Using stream api  
 List*<EmployeePredicate> youngEmployees = employees.stream().filter(e -> e.getAge()>21 && e.getAge()<40).collect(Collectors.*toList*());  
 System.*out*.println("Filtered young employees with stream ::: "+youngEmployees);  
 System.*out*.println("=========================================================================");  
  
 *//Without stream api using only for each loop and creating different predicates  
 Predicate*<EmployeePredicate> youngEmp = e -> e.getAge() > 21 && e.getAge() < 40;  
  
 *for*(EmployeePredicate employeePredicate : employees) {  
 *if*(youngEmp.test(employeePredicate)) {  
 System.*out*.println("Young employees without stream ::: "+employeePredicate);  
 }  
 }  
 System.*out*.println("=========================================================================");  
  
 *Predicate*<EmployeePredicate> femaleEmp = e -> e.getGender().equals("F");  
 *for*(EmployeePredicate employeePredicate : employees) {  
  
 *if*(femaleEmp.test(employeePredicate)) {  
 System.*out*.println("Female employees without stream ::: "+employeePredicate);  
 }  
 }  
 System.*out*.println("=========================================================================");  
  
 *//BiPredicate is a different type of predicate which has two arguments. First arg is class and second arg is data type for dynamic value.  
 BiPredicate*<EmployeePredicate,Integer> biPredicate = (e,age) -> e.getAge() > age;  
 *for*(EmployeePredicate employeePredicate : employees) {  
  
 *if*(biPredicate.test(employeePredicate, 35)) {  
 System.*out*.println("Using BiPredicate filter to get those employees who's greater than 35 years age ::: "+employeePredicate);  
 }  
 }  
 }  
}

**Output :-**

All employees ::: [EmployeePredicate{id=1, age=23, gender='M', firstName='Ricky', lastName='Luther'}, EmployeePredicate{id=2, age=22, gender='F', firstName='Rosy', lastName='Taylor'}, EmployeePredicate{id=3, age=33, gender='M', firstName='Jayden', lastName='West'}, EmployeePredicate{id=4, age=21, gender='M', firstName='Shane', lastName='Lawmen'}, EmployeePredicate{id=5, age=19, gender='F', firstName='Cristine', lastName='Josie'}, EmployeePredicate{id=6, age=25, gender='M', firstName='David', lastName='Feezor'}, EmployeePredicate{id=7, age=38, gender='F', firstName='Scarlet', lastName='Ellie'}, EmployeePredicate{id=8, age=41, gender='M', firstName='Russel', lastName='Gussin'}, EmployeePredicate{id=9, age=35, gender='F', firstName='Rakul', lastName='Singh'}, EmployeePredicate{id=10, age=45, gender='M', firstName='Aiden', lastName='Taylor'}]

=========================================================================

Filtered young employees with stream ::: [EmployeePredicate{id=1, age=23, gender='M', firstName='Ricky', lastName='Luther'}, EmployeePredicate{id=2, age=22, gender='F', firstName='Rosy', lastName='Taylor'}, EmployeePredicate{id=3, age=33, gender='M', firstName='Jayden', lastName='West'}, EmployeePredicate{id=6, age=25, gender='M', firstName='David', lastName='Feezor'}, EmployeePredicate{id=7, age=38, gender='F', firstName='Scarlet', lastName='Ellie'}, EmployeePredicate{id=9, age=35, gender='F', firstName='Rakul', lastName='Singh'}]

=========================================================================

Young employees without stream ::: EmployeePredicate{id=1, age=23, gender='M', firstName='Ricky', lastName='Luther'}

Young employees without stream ::: EmployeePredicate{id=2, age=22, gender='F', firstName='Rosy', lastName='Taylor'}

Young employees without stream ::: EmployeePredicate{id=3, age=33, gender='M', firstName='Jayden', lastName='West'}

Young employees without stream ::: EmployeePredicate{id=6, age=25, gender='M', firstName='David', lastName='Feezor'}

Young employees without stream ::: EmployeePredicate{id=7, age=38, gender='F', firstName='Scarlet', lastName='Ellie'}

Young employees without stream ::: EmployeePredicate{id=9, age=35, gender='F', firstName='Rakul', lastName='Singh'}

=========================================================================

Female employees without stream ::: EmployeePredicate{id=2, age=22, gender='F', firstName='Rosy', lastName='Taylor'}

Female employees without stream ::: EmployeePredicate{id=5, age=19, gender='F', firstName='Cristine', lastName='Josie'}

Female employees without stream ::: EmployeePredicate{id=7, age=38, gender='F', firstName='Scarlet', lastName='Ellie'}

Female employees without stream ::: EmployeePredicate{id=9, age=35, gender='F', firstName='Rakul', lastName='Singh'}

=========================================================================

Using BiPredicate filter to get those employees who's greater than 35 years age ::: EmployeePredicate{id=7, age=38, gender='F', firstName='Scarlet', lastName='Ellie'}

Using BiPredicate filter to get those employees who's greater than 35 years age ::: EmployeePredicate{id=8, age=41, gender='M', firstName='Russel', lastName='Gussin'}

Using BiPredicate filter to get those employees who's greater than 35 years age ::: EmployeePredicate{id=10, age=45, gender='M', firstName='Aiden', lastName='Taylor'}

**BiFunction Functional Interface :-** *java.util.function.Function*  is a functional interface whose functional method (single abstract method) is *R apply(T t,U u)*.

It takes two arguments (object of type T and U) and returns an object (object of type R). Both argument and output can be a different type.

*package* java8.functionInterface;  
*import* java.util.function.*BiFunction*;  
  
*public class* BiFunctionInterface {  
   
 *public static void* main(String[] args) {  
 *BiFunction*<Integer,Double,Boolean> bifunction = (inte,doub) -> {  
 Integer age = inte + doub.intValue();  
 Boolean result = age > 18 ? Boolean.*TRUE* : Boolean.*FALSE*;  
 *return* result;  
 };  
 System.*out*.println("BiFunction -> "+bifunction.apply(12,12.1));  
 }  
}

**Output :-** BiFunction -> true

**BiConsumer Functional Interface :-** BiConsumer Interface accepts two input arguments and does not return any result. This is the two-arity specialization of Consumer interface. It provides a functional method accept(Object, Object) to perform custom operations.

*package* java8.Consumer;  
*import* java.util.function.*BiConsumer*;  
  
*public class* BiConsumerInterface {  
   
 *public static void* main(String[] args) {  
 *BiConsumer*<Integer, Integer> addTwo = (x, y) -> System.*out*.println(x + y);  
 addTwo.accept(1, 2);  
 }  
}

**Output :-** 3

**ForEach Loop :-** In Java 8, we can use the new forEach to loop or iterate a Map, List, Set, or Stream. In the below example forEach method from Map interface, So it takes BiConsumer(two input) as input in the method. But forEach method for Stream(Single input) interface has Consumer Interface.

*package* java8.foreach;  
*import* java.util.HashMap;  
*import* java.util.*Map*;  
  
*public class* Java8ForEachLoop {  
  
 *public static void* main(String[] args) {  
 *Map*<String, Integer> map = *new* HashMap<>();  
 map.put("A", 10);  
 map.put("B", 20);  
 map.put("C", 30);  
 map.put("D", 40);   
 map.forEach((k, v) -> System.*out*.println("Key : " + k + ", Value : " + v));  
 }  
}

**Output :-**

Key : A, Value : 10

Key : B, Value : 20

Key : C, Value : 30

Key : D, Value : 40

**Note :-** The forEach does not guarantee the stream’s encounter order, regardless of whether the stream is sequential or parallel. The result is obvious when run in a parallel mode.

*package* java8.foreach;  
*import* java.util.stream.*Stream*;  
  
*public class* ParrallelForEach {  
  
 *public static void* main(String[] args) {  
 *Stream*<String> s = *Stream*.*of*("a", "b", "c", "1", "2", "3");  
 s.parallel().forEach(x -> System.*out*.println(x));  
 }  
}

**Output :-**

1

2

b

c

3

a

**ForEachOrdered Loop :-** The forEachOrdered guarantees the stream’s encounter order; thus, it sacrifices the benefit of parallelism.

*package* java8.foreach;  
*import* java.util.stream.*Stream*;  
  
*public class* ParrallelForEach {  
  
 *public static void* main(String[] args) {  
 *Stream*<String> s = *Stream*.*of*("a", "b", "c", "1", "2", "3");  
 s.parallel().forEachOrdered(x -> System.*out*.println(x));  
 }  
}

**Output :-**

a

b

c

1

2

3

**BinaryOperator Interface :-** The BinaryOperator Interface<T> is a part of the java.util.function package which has been introduced since Java 8, to implement functional programming in Java. It represents a binary operator which takes two operands and operates on them to produce a result. However, what distinguishes it from a normal BiFunciton is that both of its arguments and its return type are same. The BinaryOperator<T> extends the BiFunction<T, T, T> type.

This functional interface which takes in one generic namely:-

* **T**: denotes the type of the input arguments and the return value of the operation

*package* java8.BinaryOperator;  
*import* java.util.function.*BiFunction*;  
*import* java.util.function.*BinaryOperator*;  
  
*public class* Java8BinaryOperator1 {  
  
 *public static void* main(String[] args) {  
 *// BiFunction  
 BiFunction*<Integer, Integer, Integer> func = (x1, x2) -> x1 + x2;  
 Integer result = func.apply(2, 3);  
 System.*out*.println(result); *// 5  
  
 // BinaryOperator  
 BinaryOperator*<Integer> func2 = (x1, x2) -> x1 + x2;  
 Integer result2 = func.apply(2, 3);  
 System.*out*.println(result2); *// 5* }  
}

**Output :-**

5

5

**BinaryOperator as Argument :-** This example simulates a stream.reduce() to sum all the Integer.

*package* java8.BinaryOperator;  
*import* java.util.Arrays;  
*import* java.util.*List*;  
*import* java.util.function.*BinaryOperator*;  
  
*public class* Java8BinaryOperator2 {  
 *public static void* main(String[] args) {  
 Integer[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
 Integer result = *math*(Arrays.*asList*(numbers), 0, (a, b) -> a + b);  
 System.*out*.println(result); *// 55* Integer result2 = *math*(Arrays.*asList*(numbers), 0, Integer::*sum*);  
 System.*out*.println(result2); *// 55* }  
  
 *public static* <T> T math(*List*<T> list, T init, *BinaryOperator*<T> accumulator) {  
 T result = init;  
 *for* (T t : list) {  
 result = accumulator.apply(result, t);  
 }  
 *return* result;  
 }  
}

**Output :-**

55

55

**Note :-** If the math operations involve primitive types like int, change to IntBinaryOperator for better performance. Some minor changes have done in the above math method for same output.

*public static int* math(*int*[] list, *int* init, *IntBinaryOperator* accumulator) {  
 *int* result = init;  
 *for* (*int* t : list) {  
 result = accumulator.applyAsInt(result, t);  
 }  
 *return* result;  
}

**MaxBy and MinBy Function in BinaryOperator :-** This example uses BinaryOperator and a custom Comparator to find the highest and lowest pay developer from a list of developers.

*package* java8.BinaryOperator;  
*import* java.math.BigDecimal;  
*import* java.util.Arrays;  
*import* java.util.*Comparator*;  
*import* java.util.*List*;  
*import* java.util.function.*BinaryOperator*;  
  
*public class* Java8BinaryOperator4 {  
  
 *public static void* main(String[] args) {  
  
 Developer dev1 = *new* Developer("jordan", BigDecimal.*valueOf*(99999));  
 Developer dev2 = *new* Developer("jack", BigDecimal.*valueOf*(88888));  
 Developer dev3 = *new* Developer("jaden", BigDecimal.*valueOf*(100000));  
 Developer dev4 = *new* Developer("ali", BigDecimal.*valueOf*(30000));  
 Developer dev5 = *new* Developer("mkyong", BigDecimal.*valueOf*(25000));  
  
 *List*<Developer> list = Arrays.*asList*(dev1, dev2, dev3, dev4, dev5);  
  
 *// 1. Create a Comparator  
 Comparator*<Developer> comparing = *Comparator*.*comparing*(Developer::getSalary);  
  
 *// 2. BinaryOperator with a custom Comparator  
 BinaryOperator*<Developer> bo = *BinaryOperator*.*maxBy*(comparing);  
 Developer result = *find*(list, bo);  
 System.*out*.println(result); *// Developer{name='jaden', salary=10000}  
  
 // one line  
 // find developer with highest pay* Developer developer = *find*(list, *BinaryOperator*.*maxBy*(*Comparator*.*comparing*(Developer::getSalary)));  
 System.*out*.println(developer); *// Developer{name='jaden', salary=100000}  
  
 // find developer with lowest pay* Developer developer2 = *find*(list, *BinaryOperator*.*minBy*(*Comparator*.*comparing*(Developer::getSalary)));  
 System.*out*.println(developer2); *// Developer{name='mkyong', salary=25000}* }  
  
 *public static* Developer find(*List*<Developer> list, *BinaryOperator*<Developer> accumulator) {  
 Developer result = *null*;  
 *for* (Developer t : list) {  
 *if* (result == *null*) {  
 result = t;  
 } *else* {  
 result = accumulator.apply(result, t);  
 }  
 }  
 *return* result;  
 }  
}

**Output :-**

Developer{name='jaden', salary=100000}

Developer{name='jaden', salary=100000}

Developer{name='mkyong', salary=25000}

**UnaryOperator Interface :-** In Java 8, UnaryOperator is a functional interface and it extends Function. The UnaryOperator takes one argument, and returns a result of the same type of its arguments.

@FunctionalInterface

public interface UnaryOperator <T> extends Function <T,T> { }

The Function takes one argument of any type and returns a result of any type.

@FunctionalInterface

public interface Function <T, R> {

R apply(T t);

}

*package* java8.UnaryOperator;  
*import* java.util.ArrayList;  
*import* java.util.Arrays;  
*import* java.util.*List*;  
*import* java.util.function.*UnaryOperator*;  
  
*public class* Java8UnaryOperator2 {  
  
 *public static void* main(String[] args) {  
 *List*<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
  
 *List*<Integer> result = *math*(list, x -> x \* 2);  
 System.*out*.println(result); *// [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]* }  
  
 *public static* <T> *List*<T> math(*List*<T> list, *UnaryOperator*<T> uo) {  
 *List*<T> result = *new* ArrayList<>();  
 *for* (T t : list) {  
 result.add(uo.apply(t));  
 }  
 *return* result;  
 }  
}

**Output :-** [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

**Arrays Stream :-** The stream() method of the Arrays class in Java is a utility method that allows you to obtain a sequential stream of elements from an array. It was introduced in Java 8 as part of the Stream API.

The stream() method is overloaded for different types of arrays, such as arrays of primitive types (int[], double[], etc.) and arrays of reference types (Object[]). When invoked on an array, it returns a sequential stream that represents the elements of the array as individual stream elements.

### Syntax of Array Stream() –

### static datatypeStream stream(datatype[] array)

Here, the array represents the array from which you want to obtain the stream. The method returns a Stream object that you can use to perform various operations on the array elements, such as filtering, mapping, or reducing.

The stream() method is commonly used in combination with other stream operations to process arrays in a functional programming style, enabling concise and expressive code for array manipulation and processing.

*package* java8.stream.ArrayStream;  
*import* java.util.Arrays;  
*import* java.util.stream.*IntStream*;  
*import* java.util.stream.*Stream*;  
*class* StudyTonight{  
 *public static void* main(String args[])  
 {  
 String[] stringArray = { "java", "cpp", "c", "python" };  
 *Stream*<String> stringStream = Arrays.*stream*(stringArray);  
 stringStream.forEach(str -> System.*out*.print(str + " "));  
  
 System.*out*.println();  
  
 *int*[] intArray = {12, 41, 18, 4, 5, 31};  
 *IntStream* intStream = Arrays.*stream*(intArray);  
 intStream.forEach(str -> System.*out*.print(str + " "));  
 }  
}

**Output :-**

java cpp c python

12 41 18 4 5 31

### Stream API :- A Stream in Java can be defined as a sequence of elements from a source. The source of elements here refers to a Collection or Array that provides data to the Stream.

* Java streams are designed in such a way that most of the stream operations (called **intermediate operations**) return a Stream. This helps to create a chain of stream operations. This is called a **stream pipeline**.
* Java streams also support the **aggregate or terminal operations** on the elements. The aggregate operations are operations that allow us to express common manipulations on stream elements quickly and clearly, for example, finding the max or min element, finding the first element matching giving criteria, and so on.
* Not that a ***stream maintains the same ordering* of the elements *as the ordering in the stream source***.

### What is a Stream :- All of us have watched online videos on YouTube. When we start watching a video, a small portion of the video file is first loaded into our computer and starts playing. we don’t need to download the complete video before we start watching it. This is called video streaming. At a very high level, we can think of the small portions of the video file as a stream and the whole video as a Collection.

### At the granular level, the difference between a Collection and a Stream is when the things are computed. A **Collection is an in-memory data structure that holds all the data structure’s values**. Every element in the Collection has to be computed before it can be added to the Collection. While a **Stream is conceptually a pipeline in which elements are computed on demand**.

### In Java, java.util.Stream interface represents a stream on which one or more operations can be performed.

* Stream operations are either intermediate or terminal. The terminal operations return a result of a certain type, and intermediate operations return the stream itself so we can chain multiple methods in a row to perform the operation in multiple steps.
* Streams are created on a source, e.g. a java.util.Collection like List or Set. The Map is not supported directly, we can create a stream of map keys, values or entries.
* Stream operations can either be executed sequentially or in parallel. when performed parallelly, it is called a *parallel stream*.

### Creating Streams :- There are five ways to create stream object.

### Stream.of() – The Stream.of method is used to create a stream from one or more elements. It takes one or more elements as arguments and returns a stream containing those elements.

### In the given example, we are creating a stream of a fixed number of integers.

Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);

### stream.forEach(p -> System.out.println(p));

### Stream.of(array) – In the given example, we are creating a stream from the array. The elements in the stream are taken from the array.

Stream<Integer> stream = Stream.of(**new** Integer[]{1,2,3,4,5 } );

### stream.forEach(p -> System.out.println(p));

### List.stream() – In the given example, we are creating a stream from the List. The elements in the stream are taken from the List.

List<Integer> list = **new** ArrayList<Integer>();

**for**(**int** i = 1; i< 10; i++){

list.add(i);

}

Stream<Integer> stream = list.stream();

### stream.forEach(p -> System.out.println(p));

### Stream.generate() – In the given example, we are creating a stream from generated elements. This will produce a stream of 5 random numbers. We have restricted the elements count using limit() function.

*package* java8.stream;  
*import* java.util.Random;  
*import* java.util.stream.*Stream*;  
  
*public class* StreamGenerate {  
 *public static void* main(String[] args) {  
 *Stream*<Integer> randomNumbers = *Stream* .*generate*(() -> (*new* Random()).nextInt(100));  
  
 randomNumbers.limit(5).forEach(n -> System.*out*.print(n+" "));  
 }  
}

**Output :-** 59 63 5 75 31 // Random numbers

### Stream.iterate() – In Java 8, we can use Stream.iterate to create stream values on demand, so called infinite stream.

*package* java8.stream;  
*import* java.util.stream.*Stream*;  
  
*public class* StreamIterate {  
 *public static void* main(String[] args) {  
 *Stream*.*iterate*(1,element -> element + 1)  
 .filter(element -> element % 10 == 0).limit(5)  
 .forEach(element -> System.*out*.print(element+" "));  
 }  
}

### Output :- 10 20 30 40 50

### Stream of String chars – In the given example, first, we create a stream from the characters of a given string. In the second part, we are creating the stream of tokens received from splitting from a string.

*package* java8.stream;  
*import* java.util.stream.*IntStream*;  
  
*public class* StreamChars {  
 *public static void* main(String[] args) {  
 *IntStream* stream = "12345\_abcdefg".chars();  
 stream.forEach(p -> System.*out*.print(p+" "));  
 }  
}

### Output :- 49 50 51 52 53 95 97 98 99 100 101 102 103 // ASCII charactors

1. **Stream Collectors :-** After performing the intermediate operations on elements in the stream, we can collect the processed elements again into a Collection using the stream Collector methods.
   1. **Collect Stream Elements to a list -** In the given example, first, we create a stream on integers 1 to 10. Then we process the stream elements to find all even numbers. At last, we are collecting all even numbers into a List.

List<Integer> list = **new** ArrayList<Integer>();

**for**(**int** i = 1; i< 10; i++){

list.add(i);

}

Stream<Integer> stream = list.stream();

List<Integer> evenNumbersList = stream.filter(i -> i%2 == 0)

.collect(Collectors.toList());

System.out.print(evenNumbersList);

**3.2 Collect Stream Elements to an Array -** The given example is similar to the first example shown above. The only difference is that we are collecting even numbers in an Array.

List<Integer> list = **new** ArrayList<Integer>();

**for**(**int** i = 1; i< 10; i++){

list.add(i);

}

Stream<Integer> stream = list.stream();

Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::**new**);

System.out.print(evenNumbersArr);

There are plenty of other ways also to collect stream into a Set, Map or into multiple ways. Just go through Collectors class and try to keep them in mind.

1. **Stream Operations :-** Stream operations has a long list of useful functions. Let us look at a few of them. Before moving ahead, let us build a List of strings beforehand. We will build our examples on this list so that it is easy to relate and understand.

*List*<String> memberNames = *new* ArrayList<>();  
 memberNames.add("Manish");  
 memberNames.add("Jyoti");  
 memberNames.add("Arti");  
 memberNames.add("Ananya");  
 memberNames.add("Chotu");  
 memberNames.add("Shikha");  
 memberNames.add("Mona");  
 memberNames.add("Sourabh");  
 memberNames.add("Diksha");

These core methods have been divided into 2 parts given below:

* 1. **Intermediate Operations -** Intermediate operations return the stream itself so you can chain multiple methods calls in a row. Let’s learn important ones.

**4.1.1. Stream.filter() - Stream filter(Predicate predicate)** returns a stream consisting of the elements of this stream that match the given predicate. This is an ***intermediate operation.*** These operations are always lazy i.e, executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.

memberNames.stream().filter((s) -> s.startsWith("S"))  
 .forEach(System.*out*::println);

**Output :-**

**Shikha**

**Sourabh**

**4.1.2. Stream.map() - Stream map(Function mapper)** returns a stream consisting of the results of applying the given function to the elements of this stream.

Stream map(Function mapper) is an **intermediate operation**. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

memberNames.stream().filter((s) -> s.startsWith("S"))  
 .map(String::toUpperCase)  
 .forEach(System.*out*::println);

**Output :-**

**SHIKHA**

**SOURABH**

**4.1.3. Stream.flatMap() -** In [Java](https://www.javatpoint.com/java-tutorial) 8 Streams, the flatMap() method applies operation as a mapper function and provides a stream of element values. It means that in each iteration of each element the map() method creates a separate new stream. By using the flattening mechanism, it merges all streams into a single resultant stream. In short, it is used to convert a Stream of Stream into a list of values.

**Syntax:** <R> Stream<R> flatMap(Function<? **super** T,? **extends** Stream<? **extends** R>> mapper)

The method takes a function as an argument. It accepts T as a parameter and returns a stream of R.

**R:** It is a type parameter that represents the element type of the new stream.

**mapper:** It is a parameter that is a non-interfering, stateless function to apply to each element. It produces a stream of new values.

*List*<String> newMembers = Arrays.*asList*("Abhi","Soumya","Bitto");  
*List*<*List*<String>> membersNames = Arrays.*asList*(memberNames,newMembers);  
*List*<String> flatMapList = membersNames.stream().flatMap(map -> map.stream()).collect(Collectors.*toList*());  
System.*out*.println("flatMapList :: "+flatMapList);

**Output :-** flatMapList :: [Manish, Jyoti, Arti, Ananya, Chotu, Shikha, Mona, Sourabh, Diksha, Abhi, Soumya, Bitto]

As already discussed in the post that flatMap() is the combination of a map and a flat operation i.e, it first applies map function and than flattens the result.

**4.1.4. Stream.sorted() -** Since Java 8, the sorted() method is part of the Stream API and is used to sort the elements of a stream. By default, elements are sorted in the natural order, but we can apply for a custom order using a Comparator.

**The Stream interface provides two methods for sorting the elements:**

* **sorted()** – Provides the default sorting.
* **sorted(Comparator)** – Sorting based on the provided comparator.

**Stream Sorted() –**

* sorted() is a stateful intermediate operation that returns a new Stream.
* It returns a stream consisting of the elements of this stream, sorted according to the natural order.
* If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
* For ordered streams, the sort is stable.
* For unordered streams, no stability guarantees are made.

*package* java8.stream;  
*import* java.util.Arrays;  
*import* java.util.*List*;  
*import* java.util.stream.Collectors;  
  
*public class* StreamSortMethod {  
  
 *public static void* main(String[] args) {  
 *List*<String> names = Arrays.*asList*("Mona","Rakhi","Anjali","Shikha","Ranu","Diksha");  
 System.*out*.println("Before Sorting ::: "+names);  
 *//sorted in stream api is sorted collection according to natural order.  
 List*<String> sortedNames = names.stream().sorted().collect(Collectors.*toList*());  
 System.*out*.println("After Sorting ::: "+sortedNames);  
 }  
}

**Output :-**

Before Sorting ::: [Mona, Rakhi, Anjali, Shikha, Ranu, Diksha]

After Sorting ::: [Anjali, Diksha, Mona, Rakhi, Ranu, Shikha]

**Stream sorted(Comparator) :-**

* This is a stateful intermediate operation that returns a new stream.
* It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
* For ordered streams, the sort is stable.
* For unordered streams, no stability guarantees are made.

*package* java8.stream;  
*import* java.util.Arrays;  
*import* java.util.*List*;  
*import* java.util.stream.Collectors;  
  
*public class* StreamSortingUsingComparator {  
  
 *public static void* main(String[] args) {  
 *List*<String> names = Arrays.*asList*("Mona","Rakhi","Anjali","Shikha","Ranu","Diksha");  
 System.*out*.println("Before Sorting ::: "+names);

*//If we compare s1 object to s2 (s1.compareTo(s2)) then it will sorted according to natural sorting order(Ascending order)*

*//If we want to sort descending order then compare s2 object to s1 (s2.compareTo(s1)).*

*List*<String> sortedNames = names.stream()  
 .sorted((s1, s2) -> s2.compareTo(s1)).collect(Collectors.*toList*());  
 System.*out*.println("After Sorting ::: "+sortedNames);  
 }  
}

**Output :-**

Before Sorting ::: [Mona, Rakhi, Anjali, Shikha, Ranu, Diksha]

After Sorting ::: [Shikha, Ranu, Rakhi, Mona, Diksha, Anjali]

**4.1.5. Stream.distinct() - The Stream.distinct() method returns a new Stream consisting of the distinct elements from the given Stream.** The distinct() operation removes duplicate elements from a stream, ensuring that only unique elements are retained in the resulting stream. The distinct() is one such **stateful intermediate operation** that uses the state from previously seen elements from the Stream while processing the new items.

* The distinct() returns the distinct elements from the given stream. For checking the equality of the stream elements, the [equals()](https://howtodoinjava.com/java/basics/java-hashcode-equals-methods/) method is used.
* The distinct() guarantees the ordering for the streams backed by an ordered collection. The element appearing first in the encounter order is preserved for ordered streams.
* For *unordered streams*, no stability guarantees are made.