* Java Nashorn

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.

You can execute JavaScript code by using jjs command-line tool and by embedding into Java source code.

Example: Executing by Using Terminal

Following is the step by step process to execute JavaScript code at the JVM.

1) Create a file hello.js.

2) Write and save the following code into the file.

1. var hello = function(){
2. print("Hello Nashorn");
3. };
4. hello();

3) Open terminal

4) Write command **jjs hello.js** and press enter.

After executing command, you will see the below output.

Output:

Hello Nashorn

Example: Executing JavaScript file in Java Code

You can execute JavaScript file directly from your Java file. In the following code, we are reading a file hello.js with the help of FileReader class.

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. // Reading Nashorn file
8. ee.eval(**new** FileReader("js/hello.js"));
9. }
10. }

Output:

Hello Nashorn

Example: Embedding JavaScript Code in Java Source File

You can embed your JavaScript code in Java source file. Java compiler will not complaint but it is not good practice when you have large source code. In the following example, we are evaluating JavaScript code.

1. **import** javax.script.\*;
2. **public** **class** NashornExample {
3. **public** **static** **void** main(String[] args) **throws** Exception{
4. // Creating script engine
5. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
6. // Evaluating Nashorn code
7. ee.eval("print('Hello Nashorn');");
8. }
9. }

Output:

Hello Nashorn

Example: Embedding JavaScript Expression

You can embed JavaScript expressions and variables in JavaScript code. In the following code we are embedding a variable to string. To execute this program you need to pass a flag -scripting in command-line.

*File: hello.js*

1. var hello = function(msg){
2. print("Hello ${msg}");
3. };
4. hello("Nashron");

**Command:** jjs -scripting hello.js

Output:

Hello Nashorn

Heredocs

In Nashorn, heredocs are simply multi-line strings. You can create it with << followed by a special termination marker, which is EOF. You can also embed JavaScript expressions in ${...} expressions.

Example : Heredocs in JavaScript File

*file: hello.js*

1. var message = <<EOF
2. This is a java script file
3. it contains multiple lines
4. of code.
5. let's execute.
6. EOF
7. print(message)

**Command:** jjs -scripting hello.js

Output:

This is a java script file

it contains multiple lines

of code.

let's execute.

Example: Setting JavaScript variable in Java File

You can pass value to JavaScript variable in the Java file. In the followed example, we are binding and passing variable to JavaScript file.

*File: hello.js*

1. print("Hello "+name);

*File: NashornExample.java*

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. //Binding script and Define scope of script
8. Bindings bind = ee.getBindings(ScriptContext.ENGINE\_SCOPE);
9. bind.put("name", "Nashorn");
10. // Reading Nashorn file
11. ee.eval(**new** FileReader("js/hello.js"));
12. }
13. }

Output:

Hello Nashorn

Import Java Package in JavaScript File

Java provides a facility to import Java package inside the JavaScript code. Here, we are using two approaches to import Java packages.

Example1: Import Java Package in JavaScript File

*File: hello.js*

1. print(java.lang.Math.sqrt(4));

Output:

2

Example2: Import Java Package in JavaScript File

*File: hello.js*

1. var importFile = **new** JavaImporter(java.util);
2. var a = **new** importFile.ArrayList();
3. a.add(12);
4. a.add(20);
5. print(a);
6. print(a.getClass());

Output:

[12, 20]

class java.util.ArrayList

Example3: Import Java Package in JavaScript File

you can import multiple packages at the same time.

*File: hello.js*

1. var importIt = **new** JavaImporter(java.lang.String,java.util,java.io);
2. with (importIt) {
3. var linkedHS = **new** LinkedHashSet();
4. linkedHS.add(**new** File("abc"));
5. linkedHS.add(**new** File("hello.js"));
6. linkedHS.add("india".toUpperCase());
7. }
8. print(linkedHS);

Output:

[abc, hello.js, INDIA]

Calling JavaScript function inside Java code

You can call JavaScript function inside the Java file. In the followed example, we are calling JavaScript functions.

Example: Calling function inside Java code

*File: hello.js*

1. var functionDemo1 = function(){
2. print("This is JavaScript function");
3. }
4. var functionDemo2 = function(message){
5. print("Hello "+message);
6. }

*File: NashornExample.java*

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. // Reading Nashorn file
8. ee.eval(**new** FileReader("js/hello.js"));
9. Invocable invocable = (Invocable)ee;
10. // calling a function
11. invocable.invokeFunction("functionDemo1");
12. // calling a function and passing variable as well.
13. invocable.invokeFunction("functionDemo2","Nashorn");
14. }
15. }

Output:

This is JavaScript function

Hello Nashorn

* JDBC Improvements

#### 1) The JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

#### 2) Added some new features in JDBC 4.2.

Java JDBC 4.2 introduces the following features:

* Addition of REF\_CURSOR support.
* Addition of java.sql.DriverAction Interface
* Addition of security check on deregisterDriver Method in DriverManager Class
* Addition of the java.sql.SQLType Interface
* Addition of the java.sql.JDBCType Enum
* Add Support for large update counts
* Changes to the existing interfaces
* Rowset 1.2: Lists the enhancements for JDBC RowSet.

Java JDBC DriverAction

It is an interface that must be implemented when a Driver wants to be notified by DriverManager. It is added in java.sql package and contains only one abstract method.

### DriverAction Method

|  |  |
| --- | --- |
| **Method** | **Description** |
| void deregister() | This method called by DriverManager.deregisterDriver(Driver) to notify the JDBC driver that it was de-registered. |

The deregister method is intended only to be used by JDBC Drivers and not by applications.

JDBC drivers are recommended not to implement the DriverAction in a public class.

If there are active connections to the database at the time that the deregister method is called, it is implementation specific as to whether the connections are closed or allowed to continue. Once this method is called, it is implementation specific as to whether the driver may limit the ability to create new connections to the database, invoke other Driver methods or throw a SQLException.

## Java JDBC4.2 DriverAction Example

1. **import** java.sql.\*;
2. // implementing DriverAction interface
3. **class** JdbcExample **implements** DriverAction{
4. // implementing deregister method of DriverAction interface
5. @Override
6. **public** **void** deregister() {
7. System.out.println("Driver deregistered");
8. }
9. **public** **static** **void** main(String args[]){
10. **try**{
11. // Creating driver instance
12. Driver driver = **new** com.mysql.jdbc.Driver();
13. // Creating Action Driver
14. DriverAction da = **new** JdbcExample();
15. // Registering driver by passing driver and driverAction
16. DriverManager.registerDriver(driver, da);
17. // Creating connection
18. Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","mysql");
19. //Here student is database name, root is username and password is mysql
20. Statement stmt=con.createStatement();
21. // Executing SQL query
22. ResultSet rs=stmt.executeQuery("select \* from user");
23. **while**(rs.next()){
24. System.out.println(rs.getInt(1)+""+rs.getString(2)+""+rs.getString(3));
25. }
26. // Closing connection
27. con.close();
28. // Calling deregisterDriver method
29. DriverManager.deregisterDriver(driver);
30. }**catch**(Exception e){ System.out.println(e);}
31. }
33. }

Output:

1 Arun 25

2 irfan 22

3 Neraj kumar 25

Driver deregistered

## Java JDBC SQLType

This interface is used to identify a generic SQL type, JDBC type or a vendor specific data type.

It provides following methods.

|  |  |
| --- | --- |
| **Method** | **Description** |
| String getName() | It returns the SQLType name that represents a SQL data type. |
| String getVendor() | It returns the name of the vendor that supports this data type. The value returned typically is the package name for this vendor. |
| Integer getVendorTypeNumber() | It returns the vendor specific type number for the data type. |

## Java JDBCType

It is an Enumeration which defines the constants that are used to identify generic SQL types, called JDBC types. It extends java.lang.Enum and implements java.sql.SQLType.

## JDBCType Fields

The following table contains constants defined in the JDBCType.

|  |  |
| --- | --- |
| **Enum constant** | **Description** |
| public static final JDBCType ARRAY | It identifies the generic SQL type ARRAY. |
| public static final JDBCType BIGINT | It identifies the generic SQL type BIGINT. |
| public static final JDBCType BIT | It identifies the generic SQL type BIT. |
| public static final JDBCType BLOB | It identifies the generic SQL type BLOB. |
| public static final JDBCType BOOLEAN | It identifies the generic SQL type BOOLEAN. |
| public static final JDBCType CHAR | It identifies the generic SQL type CHAR. |
| public static final JDBCType CLOB | It identifies the generic SQL type CLOB. |
| public static final JDBCType DATALINK | It identifies the generic SQL type DATALINK. |
| public static final JDBCType DATE | It identifies the generic SQL type DATE. |
| public static final JDBCType DECIMAL | It identifies the generic SQL type DECIMAL. |
| public static final JDBCType DISTINCT | It identifies the generic SQL type DISTINCT. |
| public static final JDBCType DOUBLE | It identifies the generic SQL type DOUBLE. |
| public static final JDBCType FLOAT | It identifies the generic SQL type FLOAT. |
| public static final JDBCType INTEGER | It identifies the generic SQL type INTEGER. |
| public static final JDBCType JAVA\_OBJECT | It indicates that the SQL type is database-specific and gets mapped to a Java object that can be accessed via the methods getObject and setObject. |
| Public static final JDBCType LONGNVARCHAR | It identifies the generic SQL type LONGNVARCHAR. |
| public static final JDBCType NCHAR | It identifies the generic SQL type NCHAR. |
| public static final JDBCType NCLOB | It identifies the generic SQL type NCLOB. |
| public static final JDBCType NULL | It identifies the generic SQL value NULL. |
| public static final JDBCType NUMERIC | It identifies the generic SQL type NUMERIC. |
| public static final JDBCType NVARCHAR | It identifies the generic SQL type NVARCHAR. |
| public static final JDBCType OTHER | It indicates that the SQL type is database-specific and gets mapped to a Java object that can be accessed via the methods getObject and setObject. |
| public static final JDBCType REAL | It identifies the generic SQL type REAL.Identifies the generic SQL type VARCHAR. |
| public static final JDBCType REF | It identifies the generic SQL type REF. |
| public static final JDBCType REF\_CURSOR | It identifies the generic SQL type REF\_CURSOR. |
| public static final JDBCType ROWID | It identifies the SQL type ROWID. |
| public static final JDBCType SMALLINT | It identifies the generic SQL type SMALLINT. |
| public static final JDBCType SQLXML | It identifies the generic SQL type SQLXML. |
| public static final JDBCType STRUCT | It identifies the generic SQL type STRUCT. |
| public static final JDBCType TIME | It identifies the generic SQL type TIME. |
| public static final JDBCType TIME\_WITH\_TIMEZONE | It identifies the generic SQL type TIME\_WITH\_TIMEZONE. |
| public static final JDBCType TIMESTAMP | It identifies the generic SQL type TIMESTAMP. |
| public static final JDBCType TIMESTAMP\_WITH\_TIMEZONE | It identifies the generic SQL type TIMESTAMP\_WITH\_TIMEZONE. |
| public static final JDBCType TINYINT | It identifies the generic SQL type TINYINT. |
| public static final JDBCType VARBINARY | It identifies the generic SQL type VARBINARY. |
| public static final JDBCType VARCHAR | It identifies the generic SQL type VARCHAR. |

## JDBCType Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| public String getName() | It returns the SQLType name that represents a SQL data type. |
| public String getVendor() | It returns the name of the vendor that supports this data type. |
| public Integer getVendorTypeNumber() | It returns the vendor specific type number for the data type. |
| public static JDBCType valueOf(int type) | It returns the JDBCType that corresponds to the specified Types value. It throws IllegalArgumentException, if this enum type has no constant with the specified Types value. |
| public static JDBCType valueOf(String name) | It returns the enum constant of this type with the specified name. The string must match exactly an identifier used to declare an enum constant in this type. It throws IllegalArgumentException, if this enum type has no constant with the specified name. It throws NullPointerException, if the argument is null. |
| public static JDBCType[] values() | It returns an array containing the constants of this enum type, in the order they are declared. This method may be used to iterate over the constants. |

* Java 8 Streams

**Streams vs. Collections**

All of us have watch online videos on youtube or some other such website. When you start watching video, a small portion of file is first loaded into your computer and start playing. You don’t need to download complete video before start playing it. This is called streaming. I will try to relate this concept with respect to collections and differentiate with Streams.

At the basic level, the difference between Collections and Streams has to do with when things are computed. A **Collection is an in-memory data structure**, which holds all the values that the data structure currently has—every element in the Collection has to be computed before it can be added to the Collection. A **Stream is a conceptually fixed data structure, in which elements are computed on demand**. This gives rise to significant programming benefits. The idea is that a user will extract only the values they require from a Stream, and these elements are only produced—invisibly to the user—as and when required. This is a form of a producer-consumer relationship.

In java, java.util.Stream represents a stream on which one or more operations can be performed. Stream **operations are either intermediate or terminal**. While **terminal operations return a result of a certain type**, **intermediate operations return the stream itself** so you can chain multiple method calls in a row. Streams are created on a source, e.g. a java.util.Collection like lists or sets (maps are not supported). Stream operations can either be executed sequential or parallel.

Based on above points, if we list down the various characteristics of Stream, they will be as follows:

* Not a data structure
* Designed for lambdas
* Do not support indexed access
* Can easily be outputted as arrays or lists
* Lazy access supported
* Parallelizable
* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

**Java Stream Interface Methods**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| boolean allMatch(Predicate<? super T> predicate) | It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| boolean anyMatch(Predicate<? super T> predicate) | It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated. |
| static <T> Stream.Builder<T> builder() | It returns a builder for a Stream. |
| <R,A> R collect(Collector<? super T,A,R> collector) | It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning. |
| <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) | It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result. |
| static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) | It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked. |
| long count() | It returns the count of elements in this stream. This is a special case of a reduction. |
| Stream<T> distinct() | It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream. |
| static <T> Stream<T> empty() | It returns an empty sequential Stream. |
| Stream<T> filter(Predicate<? super T> predicate) | It returns a stream consisting of the elements of this stream that match the given predicate. |
| Optional<T> findAny() | It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty. |
| Optional<T> findFirst() | It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned. |
| <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) | It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper) | It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper) | It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| void forEach(Consumer<? super T> action) | It performs an action for each element of this stream. |
| void forEachOrdered(Consumer<? super T> action) | It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order. |
| static <T> Stream<T> generate(Supplier<T> s) | It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc. |
| static <T> Stream<T> iterate(T seed,UnaryOperator<T> f) | It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc. |
| Stream<T> limit(long maxSize) | It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length. |
| <R> Stream<R> map(Function<? super T,? extends R> mapper) | It returns a stream consisting of the results of applying the given function to the elements of this stream. |
| DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) | It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream. |
| IntStream mapToInt(ToIntFunction<? super T> mapper) | It returns an IntStream consisting of the results of applying the given function to the elements of this stream. |
| LongStream mapToLong(ToLongFunction<? super T> mapper) | It returns a LongStream consisting of the results of applying the given function to the elements of this stream. |
| Optional<T> max(Comparator<? super T> comparator) | It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| Optional<T> min(Comparator<? super T> comparator) | It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| boolean noneMatch(Predicate<? super T> predicate) | It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| @SafeVarargs static <T> Stream<T> of(T... values) | It returns a sequential ordered stream whose elements are the specified values. |
| static <T> Stream<T> of(T t) | It returns a sequential Stream containing a single element. |
| Stream<T> peek(Consumer<? super T> action) | It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. |
| Optional<T> reduce(BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any. |
| T reduce(T identity, BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. |
| <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) | It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. |
| Stream<T> skip(long n) | It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned. |
| Stream<T> sorted() | It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed. |
| Stream<T> sorted(Comparator<? super T> comparator) | It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator. |
| Object[] toArray() | It returns an array containing the elements of this stream. |
| <A> A[] toArray(IntFunction<A[]> generator) | It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing. |

**Different ways to build streams**

##### 1) Using Stream.of(val1, val2, val3….)

public class StreamBuilders {

     public static void main(String[] args){

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

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

     }

}

##### 2) Using Stream.of(arrayOfElements)

ublic class StreamBuilders {

     public static void main(String[] args){

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

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

     }

}

##### 3) Using someList.stream()

public class StreamBuilders {

     public static void main(String[] args){

         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));

     }

}

* Streams filter() and collect ()

Before Java 8, filter a List like this:

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class BeforeJava8 {

public static void main(String[] args) {

List<String> lines = Arrays.asList("amit", "amey", "ajit");

List<String> result = getFilterOutput(lines, "amey");

for (String temp : result) {

System.out.println(temp);

}

}

private static List<String> getFilterOutput(List<String> lines, String filter) {

List<String> result = new ArrayList<>();

for (String line : lines) {

if (!"amey".equals(line)) { // we dont like mkyong

result.add(line);

}

}

return result;

}

}

Output

amit

ajit

The equivalent example in Java 8, stream.filter() to filter a List, and  () to convert a stream into a List.

package com.mkyong.java8;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class NowJava8 {

public static void main(String[] args) {

List<String> lines = Arrays.asList("amit", "amey", "ajit");

List<String> result = lines.stream() // convert list to stream

.filter(line -> !"amey".equals(line)) // we dont like mkyong

.collect(Collectors.toList()); // collect the output and convert streams to a List

result.forEach(System.out::println); //output : spring, node

}

}

* Streams filter( ), findAny( ) and orElse( )

**Person.java**

**package** com.java8.streams;

**public** **class** Person {

**private** String name;

**private** **int** age;

**public** Person(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

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

**this**.age = age;

}

}

**BeforeJava8.java:**

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** BeforeJava8 {

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

List<Person> persons = Arrays.*asList*(

**new** Person("mkyong", 30),

**new** Person("jack", 20),

**new** Person("lawrence", 40)

);

Person result = getPersonByName (persons, "jack");

System.***out***.println(result.getName()+"\t"+result.getAge());

}

**private** **static** Person getPersonByName(List<Person> persons, String name) {

Person result = **null**;

**for** (Person temp : persons) {

**if** (name.equals(temp.getName())) {

result = temp;

}

}

**return** result;

}

}

Output

Person{name='jack', age=20}

The equivalent example in Java 8, use stream.filter() to filter a List, and .findAny().orElse (null) to return an object conditional.

**NowJava8.java:**

**package** com.java8.streams;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** NowJava8 {

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

List<Person> persons = Arrays.*asList*(

**new** Person("mkyong", 30),

**new** Person("jack", 20),

**new** Person("lawrence", 40)

);

Person result1 = persons.stream() // Convert to steam

.filter(x -> "jack".equals(x.getName())) // we want "jack" only

.findAny() // If 'findAny' then return found

.orElse(**null**); // If not found, return null

System.***out***.println(result1.getName()+"\t"+result1.getAge());

Person result2 = persons.stream()

.filter(x -> "ahmook".equals(x.getName()))

.findAny()

.orElse(**null**);

System.***out***.println(result2);

}

}

Output :

Person{name='jack', age=20}

null

**For multiple condition**

**package** com.java8.streams;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** NowJava8 {

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

List<Person> persons = Arrays.*asList*(

**new** Person("mkyong", 30),

**new** Person("jack", 20),

**new** Person("lawrence", 40)

);

Person result1 = persons.stream()

.filter((p) -> "jack".equals(p.getName()) && 20 == p.getAge())

.findAny()

.orElse(**null**);

System.***out***.println(result1.getName()+"\t"+result1.getAge());

//or like this

Person result2 = persons.stream()

.filter(p -> {

**if** ("jack".equals(p.getName()) && 20 == p.getAge()) {

**return** **true**;

}

**return** **false**;

}).findAny()

.orElse(**null**);

System.***out***.println(result2.getName()+"\t"+result2.getAge());

}

}

* Streams filter() and map()

**package** com.java8.streams;

**import** java.util.Arrays;

**import** java.util.List;

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

**public** **class** NowJava8 {

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

List<Person> persons = Arrays.*asList*(

**new** Person("mkyong", 30),

**new** Person("jack", 20),

**new** Person("lawrence", 40)

);

String name = persons.stream()

.filter(x -> "jack".equals(x.getName()))

.map(Person::getName) //convert stream to String

.findAny()

.orElse("");

System.***out***.println("name : " + name);

List<String> collect = persons.stream()

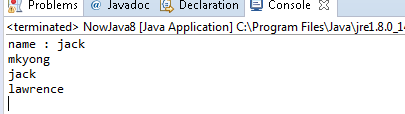
.map(Person::getName)

.collect(Collectors.*toList*());

collect.forEach(System.***out***::println);

}

}



**package** com.java8.streams;

**import** java.util.Arrays;

**import** java.util.IntSummaryStatistics;

**import** java.util.List;

**import** java.util.Random;

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

**public** **class** StreamTest {

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

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

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

System.***out***.println("Empty Strings: " + count);

count = (**int**) strings.stream().filter(string -> string.length() == 3).count();

System.***out***.println("Strings of length 3: " + count);

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

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

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

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

List<Integer> integers = Arrays.*asList*(1,2,13,4,15,6,17,8,19);

IntSummaryStatistics stats = integers.stream().mapToInt((x) ->x).summaryStatistics();

System.***out***.println("Highest number in List : " + stats.getMax());

System.***out***.println("Lowest number in List : " + stats.getMin());

System.***out***.println("Sum of all numbers : " + stats.getSum());

System.***out***.println("Average of all numbers : " + stats.getAverage());

System.***out***.println("Random Numbers: ");

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

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

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

System.***out***.println("Empty Strings: " + count);

}

}

* Java 8 Date

## Why do we need new Java Date Time API?

1. Java Date Time classes are not defined consistently, we have Date Class in both java.util as well as java.sql packages. Again formatting and parsing classes are defined in java.text package.
2. java.util.Date contains both date and time, whereas java.sql.Date contains only date. Having this in java.sql package doesn’t make sense. Also both the classes have same name, that is a very bad design itself.
3. There are no clearly defined classes for time, timestamp, formatting and parsing. We have java.text.DateFormat abstract class for parsing and formatting need. Usually SimpleDateFormat class is used for parsing and formatting.
4. All the Date classes are mutable, so they are not thread safe. It’s one of the biggest problem with Java Date and Calendar classes.
5. Date class doesn’t provide internationalization, there is no timezone support. So java.util.Calendar and java.util.TimeZone classes were introduced, but they also have all the problems listed above.

There are some other issues with the methods defined in Date and Calendar classes but above problems make it clear that a robust Date Time API was needed in Java. That’s why [Joda Time](http://www.joda.org/joda-time/" \t "_blank) played a key role as a quality replacement for Java Date Time requirements.

## Why Joda Time?

The standard date and time classes prior to Java SE 8 are poor. By tackling this problem head-on, Joda-Time became the de facto standard date and time library for Java prior to Java SE 8. Note that from Java SE 8 onwards, users are asked to migrate to java.time (JSR-310) - a core part of the JDK which replaces this project.

The design allows for multiple calendar systems, while still providing a simple API. The “default” calendar is the [ISO8601](http://www.joda.org/joda-time/cal_iso.html) standard which is used by many other standards. The Gregorian, Julian, Buddhist, Coptic, Ethiopic and Islamic calendar systems are also included. Supporting classes include time zone, duration, format and parsing.

## Java 8 Date Time API Packages

1. **java.time Package**: This is the base package of new Java Date Time API. All the major base classes are part of this package, such **as LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration** etc. All of these classes are immutable and thread safe. Most of the times, these classes will be sufficient for handling common requirements.
2. **java.time.chrono Package**: This package defines generic APIs for non ISO calendar systems. We can extend **AbstractChronology** class to create our own calendar system.
3. **java.time.format Package**: This package contains classes used for formatting and parsing date time objects. Most of the times, we would not be directly using them because principle classes in java.time package provide formatting and parsing methods.
4. **java.time.temporal Package**: This package contains temporal objects and we can use it for find out specific date or time related to date/time object. For example, we can use these to find out the first or last day of the month. You can identify these methods easily because they always have format “withXXX”.
5. **java.time.zone Package**: This package contains classes for supporting different time zones and their rules.

* Java 8 Collection API Improvements

## Sorting Map directly with Comparators.

As we know Map is in order, it is a lot of struggle to get it sorted. Now Map interface added default methods which gives you comparators for different styles like comparingByKey, comparingByValue.

Map<String, String> map = new HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.put("Z", "z");

List<Map.Entry<String, String>> sortedByKey = map.entrySet().stream().sorted(Map.Entry.comparingByKey())

.collect(Collectors.toList());

sortedByKey.forEach(System.out**::println**);

output :

B=b

C=c

Z=z

1. **Iterate over map easily with forEach**

Map<**String**, **String**> map = **new** HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.put("Z", "z");

map.forEach((k, v) -> System.out.println("Key : " + k + " Value : " + v));

* Java 8 Optional Class

***Optional is a container type for a value which may be absent***  
Consider the following function which takes a user id, fetches the user’s details with the given id from the database and returns it -

User findUserById(String userId) { ... };

If userId is not present in the database then the above function returns null. Now, let’s consider the following code written by a client -

User user = findUserById("667290");

System.out.println("User's Name = " + user.getName());

A common NullPointerException situation, right? The developer forgot to add the null check in his code. If userId is not present in the database, then the above code snippet will throw a NullPointerException.

Now, let’s understand how Optional will help you mitigate the risk of running into NullPointerException here -

Optional<User> findUserById(String userId) { ... };

By returning Optional<User> from the function, we have made it clear to the clients of this function that there might not be a User with the given userId. Now the clients of this function are **explicitly forced** to handle this fact.

The client code can now be written as -

Optional<User> optional = findUserById("667290");

optional.ifPresent(user -> {

System.out.println("User's name = " + user.getName());

})

Once you have an Optional object, you can use various utility methods to work with the Optional. The ifPresent()method in the above example calls the supplied [lambda expression](https://www.callicoder.com/java-lambda-expression-tutorial/) if the user is present, otherwise it does nothing.

Well! You get the idea here right? The client is now **forced** by the type system to write the Optional check in his code.

## Creating an Optional object

**1. Create an empty Optional**

An empty Optional Object describes the absence of a value.

Optional<User> user = Optional.empty();

**2. Create an Optional with a non-null value -**

User user = new User("667290", "Rajeev Kumar Singh");

Optional<User> userOptional = Optional.of(user);

If the argument supplied to Optional.of() is null, then it will throw a NullPointerException immediately and the Optional object won’t be created.

**3. Create an Optional with a value which may or may not be null -**

Optional<User> userOptional = Optional.ofNullable(user);

If the argument passed to Optional.ofNullable() is non-null, then it returns an Optional containing the specified value, otherwise it returns an empty Optional.

## Checking the presence of a value

**1. isPresent()**

isPresent() method returns true if the Optional contains a non-null value, otherwise it returns false.

if(optional.isPresent()) {

// value is present inside Optional

System.out.println("Value found - " + optional.get());

} else {

// value is absent

System.out.println("Optional is empty");

}

**2. ifPresent()**

ifPresent() method allows you to pass a [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) function that is executed if a value is present inside the Optional object.

It does nothing if the Optional is empty.

optional.ifPresent(value -> {

System.out.println("Value found - " + value);

});

Note that I have supplied a lambda expression to the ifPresent() method. This makes the code more readable and concise.

## Retrieving the value using get() method

Optional’s get() method returns a value if it is present, otherwise it throws [NoSuchElementException](https://docs.oracle.com/javase/8/docs/api/java/util/NoSuchElementException.html).

User user = optional.get()

**You should avoid using get() method on your Optionals without first checking whether a value is present or not, because it throws an exception if the value is absent.**

## Returning default value using orElse()

orElse() is great when you want to return a default value if the Optional is empty. Consider the following example -

// return "Unknown User" if user is null

User finalUser = (user != null) ? user : new User("0", "Unknown User");

Now, let’s see how we can write the above logic using Optional’s orElse() construct -

// return "Unknown User" if user is null

User finalUser = optionalUser.orElse(new User("0", "Unknown User"));

## Returning default value using orElseGet()

Unlike orElse(), which returns a default value directly if the Optional is empty, orElseGet() allows you to pass a [Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) function which is invoked when the Optional is empty. The result of the Supplier function becomes the default value of the Optional -

User finalUser = optionalUser.orElseGet(() -> {

return new User("0", "Unknown User");

});

## Throw an exception on absence of a value

You can use orElseThrow() to throw an exception if Optional is empty. A typical scenario in which this might be useful is - returning a custom ResourceNotFound() exception from your REST API if the object with the specified request parameters does not exist.

@GetMapping("/users/{userId}")

public User getUser(@PathVariable("userId") String userId) {

return userRepository.findByUserId(userId).orElseThrow(

() -> new ResourceNotFoundException("User not found with userId " + userId);

);

}

## Filtering values using filter() method

Let’s say you have an Optional object of User. You want to check its gender and call a function if it’s a MALE. Here is how you would do it using old school method -

if(user != null && user.getGender().equalsIgnoreCase("MALE")) {

// call a function

}

Now, let’s use Optional along with filter to achieve the same -

userOptional.filter(user -> user.getGender().equalsIgnoreCase("MALE"))

.ifPresent(() -> {

// Your function

})

The filter() method takes a predicate as an argument. If the Optional contains a non-null value and the value matches the given predicate, then filter() method returns an Optional with that value, otherwise it returns an empty Optional.

So, the function inside ifPresent() in the above example will be called if and only if the Optional contains a user and user is a MALE.

## Extracting and transforming values using map()

Let’s say that you want to get the address of a user if it is present and print it if the user is from India.

Considering the following getAddress() method inside User class -

Address getAddress() {

return this.address;

}

Here is how you would achieve the desired result -

if(user != null) {

Address address = user.getAddress();

if(address != null && address.getCountry().equalsIgnoreCase("India")) {

System.out.println("User belongs to India");

}

}

Now, let’s see how we can get the same result using map() method -

userOptional.map(User::getAddress)

.filter(address -> address.getCountry().equalsIgnoreCase("India"))

.ifPresent(() -> {

System.out.println("User belongs to India");

});

You see how concise and readable the above code is? Let’s break the above code snippet and understand it in detail -

// Extract User's address using map() method.

Optional<Address> addressOptional = userOptional.map(User::getAddress)

// filter address from India

Optional<Address> indianAddressOptional = addressOptional.filter(address -> address.getCountry().equalsIgnoreCase("India"));

// Print, if country is India

indianAddressOptional.ifPresent(() -> {

System.out.println("User belongs to India");

});

In the above example, map() method returns an empty Optional in the following cases - 1. user is absent in userOptional. 2. user is present but getAdderess() returns null.

otherwise, it returns an Optional<Address> containing user’s address.

## Cascading Optionals using flatMap()

Let’s consider the above map() example again. You might ask that if user’s address can be null then why the heck aren’t you returning an Optional<Address> instead of plain Address from getAddress() method?

And, You’re right! Let’s correct that, let’s now assume that getAddress() returns Optional<Address>. Do you think that above code will still work?

The answer is no! The problem is the following line -

Optional<Address> addressOptional = userOptional.map(User::getAddress)

Since getAddress() returns Optional<Address>, the return type of userOptional.map() will be Optional<Optional<Address>>

Optional<Optional<Address>> addressOptional = userOptional.map(User::getAddress)

Oops! We certainly don’t want that nested Optional. Let’s use flatMap() to correct that -

Optional<Address> addressOptional = userOptional.flatMap(User::getAddress)

Cool! So, Rule of thumb here - if the mapping function returns an Optional, use *flatMap()* instead of *map()* to get the flattened result from your Optional

**package** com.java8.optional;

**import** java.util.Optional;

**public** **class** OptionalBasicExample {

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

String[] str = **new** String[10];

str[5] = "JAVA OPTIONAL CLASS EXAMPLE"; // Setting value for 5th index

// It returns an empty instance of Optional class

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

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

// It returns a non-empty Optional

Optional<String> value = Optional.*of*(str[5]);

// If value is present, it returns an Optional otherwise returns an empty Optional

System.***out***.println("Filtered value: "+value.filter((s)->s.equals("Abc")));

System.***out***.println("Filtered value: "+value.filter((s)->s.equals("JAVA OPTIONAL CLASS EXAMPLE")));

// It returns value of an Optional. if value is not present, it throws an NoSuchElementException

System.***out***.println("Getting value: "+value.get());

// It returns hashCode of the value

System.***out***.println("Getting hashCode: "+value.hashCode());

// It returns true if value is present, otherwise false

System.***out***.println("Is value present: "+value.isPresent());

// It returns non-empty Optional if value is present, otherwise returns an empty Optional

System.***out***.println("Nullable Optional: "+Optional.*ofNullable*(str[5]));

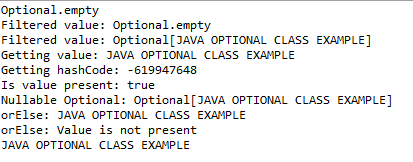
// It returns value if available, otherwise returns specified value,

System.***out***.println("orElse: "+value.orElse("Value is not present"));

System.***out***.println("orElse: "+empty.orElse("Value is not present"));

value.ifPresent(System.***out***::println); // printing value by using method reference

}

}  
  


Methods :

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static <T> Optional<T> empty() | It returns an empty Optional object. No value is present for this Optional. |
| public static <T> Optional<T> of(T value) | It returns an Optional with the specified present non-null value. |
| public static <T> Optional<T> ofNullable(T value) | It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional. |
| public T get() | If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException. |
| public boolean isPresent() | It returns true if there is a value present, otherwise false. |
| public void ifPresent(Consumer<? super T> consumer) | If a value is present, invoke the specified consumer with the value, otherwise do nothing. |
| public Optional<T> filter(Predicate<? super T> predicate) | If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional. |
| public <U> Optional<U> map(Function<? super T,? extends U> mapper) | If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. Otherwise return an empty Optional. |
| public <U> Optional<U> flatMap(Function<? super T,Optional<U> mapper) | If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. |
| public T orElse(T other) | It returns the value if present, otherwise returns other. |
| public T orElseGet(Supplier<? extends T> other) | It returns the value if present, otherwise invoke other and return the result of that invocation. |
| public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable | It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier. |
| public boolean equals(Object obj) | Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if:   * It is also an Optional and; * Both instances have no value present or; * the present values are "equal to" each other via equals(). |
| public int hashCode() | It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present. |
| public String toString() | It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions |

* Java 8 FlatMap

 Stream.flatMap() returns the stream which will contain the elements obtained by replacement of each element of the source stream by a mapping function and and flattens the result. Mapping function will produce stream and each mapped stream is closed after applying the mapping. It is useful to apply statistical function on the stream of objects and can be coded in a single line. Optional.flatMap() applies the Optional-bearing mapping function to it if value is present.

* **Stream flatMap with List**

Here we have a List of writers. Each writer has list of books. Using Stream.flatMap() we will get stream of books from all writers. And then we will find the book with highest price. We will understand it step wise.   
**1.** Stream of writers.

{

{"Mohan",

{

{10,"AAA"}, {20,"BBB"}

}

},

{"Sohan",

{

{30,"XXX"}, {15,"ZZZ"}

}

}

}

**2.** After flatMap(writer -> writer.getBooks().stream()), find the stream of books.

{

{10,"AAA"},

{20,"BBB"},

{30,"XXX"},

{15,"ZZZ"}

}

Here the result has been flattened by flatMap().   
  
**3.** After max(new BookComparator()), find the book with maximum price.

{30,"XXX"}

**FlatmapWithList.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class FlatmapWithList {

public static void main(String[] args) {

List<Book> books = Arrays.asList(new Book(10, "AAA"), new Book(20, "BBB"));

Writer w1 = new Writer("Mohan", books);

books = Arrays.asList(new Book(30, "XXX"), new Book(15, "ZZZ"));

Writer w2 = new Writer("Sohan", books);

List<Writer> writers = Arrays.asList(w1, w2);

Book book = writers.stream().flatMap(writer -> writer.getBooks().stream())

.max(new BookComparator()).get();

System.out.println("Name:"+book.getName()+", Price:"+ book.getPrice() );

}

}

**Writer.java**

package com.concretepage;

import java.util.List;

public class Writer {

private String name;

private List<Book> books;

public Writer(String name, List<Book> books) {

this.name = name;

this.books = books;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public List<Book> getBooks() {

return books;

}

public void setBooks(List<Book> books) {

this.books = books;

}

}

**Book.java**

package com.concretepage;

public class Book {

private int price;

private String name;

public Book(int price, String name) {

this.price = price;

this.name = name;

}

public int getPrice() {

return price;

}

public void setPrice(int price) {

this.price = price;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**BookComparator.java**

package com.concretepage;

import java.util.Comparator;

public class BookComparator implements Comparator<Book> {

@Override

public int compare(Book b1, Book b2) {

if (b1.getPrice() > b2.getPrice()) {

return 1;

} else if (b1.getPrice() == b2.getPrice()) {

return 0;

} else {

return -1;

}

}

}

Find the output.

Name:XXX, Price:30

Stream flatMap with List of Lists

Here we will use flatMap with list of lists. We are creating two lists and each list is containing the objects of Book. Finally I am adding these two lists in a third list. We will find out the book with minimum price.   
**FlatmapWithListOfList.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class FlatmapWithListOfList {

public static void main(String[] args) {

List<Book> list1 = Arrays.asList(new Book(10, "AAA"), new Book(20, "BBB"));

List<Book> list2 = Arrays.asList(new Book(30, "XXX"), new Book(15, "ZZZ"));

List<List<Book>> finalList = Arrays.asList(list1, list2);

Book book = finalList.stream().flatMap(list -> list.stream()).min(new BookComparator()).get();

System.out.println("Name:"+book.getName()+", Price:"+ book.getPrice() );

}

}

Find the output.

Name:AAA, Price:10

Stream flatMap with Array

Here we will use flatMap with array. I am creating a two dimensional array with integer data. Finally we will find out even numbers.   
**1.** Sample Array

{{1,2},{3,4},{5,6}}

**2.** After flatMap(row -> Arrays.stream(row))

{1,2,3,4,5,6}

**3.** After filter(num -> num%2 == 0)

{2,4,6}

Now find the example.   
**FlatMapWithArray.java**

package com.concretepage;

import java.util.Arrays;

public class FlatMapWithArray {

public static void main(String[] args) {

Integer[][] data = {{1,2},{3,4},{5,6}};

Arrays.stream(data).flatMap(row -> Arrays.stream(row)).filter(num -> num%2 == 0).

forEach(System.out::println);

}

}

Find the output.

2

4

6

Stream flatMap with Array of Objects

Here we will provide the example of flatMap with array of objects. We will create two dimensional array of Writer. This class will contain list of books. We will find the book with maximum price.   
**FlatMapWithArrayOfObject.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class FlatMapWithArrayOfObject {

public static void main(String[] args) {

List<Book> books = Arrays.asList(new Book(10, "AAA"), new Book(20, "BBB"));

Writer w1 = new Writer("Mohan", books);

books = Arrays.asList(new Book(30, "CCC"), new Book(15, "DDD"));

Writer w2 = new Writer("Sohan", books);

books = Arrays.asList(new Book(45, "EEE"), new Book(25, "FFF"));

Writer w3 = new Writer("Vikas", books);

books = Arrays.asList(new Book(5, "GGG"), new Book(15, "HHH"));

Writer w4 = new Writer("Ramesh", books);

Writer[][] writerArray = {{w1,w2},{w3,w4}};

Book book = Arrays.stream(writerArray).flatMap(row -> Arrays.stream(row)).

flatMap(writer -> writer.getBooks().stream()).max(new BookComparator()).get();

System.out.println("Name:"+book.getName()+", Price:"+ book.getPrice() );

}

}

Find the output. 

Name:EEE, Price:45

Stream flatMap with Files.lines()

Files.lines() has been introduced in Java 8. It reads all the lines of the file as a stream. Here in our example we have a file with some lines. We will store all the words in a list and print it out.   
**info.txt**

My name is Mohan

Country is India

**FlatMapWithFile.java**

package com.concretepage;

import java.io.IOException;

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class FlatMapWithFile {

public static void main(String[] args) {

Stream<String> lines = null;

try {

lines = Files.lines(Paths.get("D:/cp/info.txt"), StandardCharsets.UTF\_8);

} catch (IOException e) {

e.printStackTrace();

}

Stream<String> stream = lines.flatMap(line -> Stream.of(line.split(" +")));

List<String> words = new ArrayList<>();

stream.forEach(w->words.add(w));

words.forEach(w -> System.out.println(w));

}

}

Find the output.

My

name

is

Mohan

Country

is

India

**Optional flatMap :**

Optional has been introduced in Java 8. It behaves like a container that may keep non-null value. It handles NullPointerException. flatMap is applied only if value is present. Find the example.   
**OptionalflatMap.java**

package com.concretepage;

import java.util.Optional;

public class OptionalflatMap {

public static void main(String[] args) {

Optional<PrimeMinister> primeMinister = Optional.of(new PrimeMinister("Narendra Modi", 65));

Optional<Country> country = Optional.of(new Country(primeMinister));

Optional<Person> person = Optional.of(new Person(country));

String pmName= person.flatMap(Person::getCountry).flatMap(Country::getPrimeMinister)

.map(PrimeMinister::getName).orElse("None");

System.out.println(pmName);

}

}

**Country.java**

package com.concretepage;

import java.util.Optional;

public class Country {

Optional<PrimeMinister> primeMinister;

public Country(){}

public Country(Optional<PrimeMinister> primeMinister){

this.primeMinister = primeMinister;

}

public Optional<PrimeMinister> getPrimeMinister() {

return primeMinister;

}

public void setPrimeMinister(Optional<PrimeMinister> primeMinister) {

this.primeMinister = primeMinister;

}

}

**Person.java**

package com.concretepage;

import java.util.Optional;

public class Person {

Optional<Country> country;

public Person(){}

public Person(Optional<Country> country){

this.country = country;

}

public Optional<Country> getCountry() {

return country;

}

public void setCountry(Optional<Country> country) {

this.country = country;

}

}

Find the output.

Narendra Modi