Java8 Features

Prepare for Modularization
Type Annotations (JSR 308)
Generalized Target-Type Inference
TSL Server Name Indication
HTTP URL Permissions
DocTree API
Bulk Data Operations
Method Referance
Unicode 6.2
Enhanced Verification Errors
Parameter Names
Compact Profiles
Statically-Linked JNI Libraries
Functional Interface
Java 8
www.zishanbilal.com
Parallel Array Sorting
Streams
Lambda (JSR 335)
Limited doPrivileged
Repeating Annotations
JDBC-ODBC Bridge removal
Method Handles
Base64
Fence Intrinsics
Date/Time API (JSR 310)
Nashorn
Remove the Permanent Generation

By

Praveen Oruganti

Blog: https://praveenoruganti.blogspot.com

Facebook Group: https://www.facebook.com/groups/2340886582907696/

Github repo: https://github.com/praveenoruganti

Java 8 Features

1. Lambda Expression
2. Functional Interface
3. Method Reference
4. Stream API
5. New Date-Time API

1.Lambda Expression

It facilitates functional programming and it came up as an alternative for annonymous function.
It is applicable only for functional Interface and it reduces the boiler plate code.

```
(parameters) -> expression;
Or
(parameters) -> {statements};
```

1. **With no parameter**

```
Runnable r= () -> System.out.println("This in run method") ;
```

2. **With single parameter**

```
public Interface DisplayInterface{
void display(String name);
}
DisplayInterface di= (name) -> System.out.println(name) ;
di.display("Praveen") ;
```

3. **With multiple parameters**

```
public Interface Calculation{
void sum(int num1, int num2) ;
}
Calculation cal=(num1, num2) -> System.out.println(num1+num2) ;
cal.sum(15, 20) ;
```

2.Functional Interface

It is an Interface which has only one abstract method and can have any number of default and static methods.
It needs to be represented with the help of annotation @FunctionalInterface.

For example

```
@FunctionalInterface
public interface Display{
void display();
default void print(){
System.out.println(" This is in default method");
}
static void show() {
System.out.println("This is in static method");
}
}
```

<u>What is the need of default method in an interface?</u>
We can provide default functionality with default method.
So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

For example, i have 2 similar default methods in 2 interfaces and a class implementing those 2 interfaces will get an compile error like duplicate default methods so in order to avoid that we need to override the same method in the class as well.

For example,

```
package com.praveen.java;

public class FunctionalInterfaceTest implements Interface1, Interface2 {
@Override
public void print() {
System.out.println("Print");
}
@Override
public void show() {
System.out.println("Show");
}
@Override
public void display() {
System.out.println("Class Display");
}
}
@FunctionalInterface
interface Interface1 {
void show();
default void display() {
System.out.println("Interface1 Display");
}
}
@FunctionalInterface
```

```
interface Interface2 {
void print();
default void display() {
System.out.println("Interface2 Display");
}
}
```

What is the need of static method in an interface?

Java 8 introduced static methods to provide utility methods on interface level without creating the object of the interface.

For example,

Stream.of()

Stream.iterate()

Stream.empty()

For example,

```
package com.praveen.java;

public class FunctionalInterfaceTest implements Interface1, Interface2 {
@Override
public void print() {
System.out.println("Print");
}
@Override
public void show() {
System.out.println("Show");
}
@Override
public void display() {
System.out.println("Class Display");
}

public static void main(String[] args) {
Interface1.display1();
Interface2.display1();
}
}

@FunctionalInterface
interface Interface1 {
void show();
default void display() {
System.out.println("Interface1 Display");
}
static void display1() {
System.out.println("Interface1 Display1");
}
}

@FunctionalInterface
interface Interface2 {
void print();
default void display() {
System.out.println("Interface2 Display");
}
static void display1() {
System.out.println("Interface2 Display1");
}
}
```

Predefined Functional Interfaces

a. Predicate -> test()
b. Function -> apply()
c. Optional -> isPresent()
d. Consumer -> accept()
e. Supplier -> get()

a.Predicate

It accepts an object as input and returns a boolean i.e.. true or false.

For example,

```
Predicate<String> containsLetterP = s -> s.contains("P");
if(containsLetterP.test("Praveen")) {
System.out.println("This is Praveen");
}else {
System.out.println("No name");
}
```

Predicate chaining is also possible and we can use and(), or(),negate()

b.Function

Function is similar to Predicate but it returns an object.
Function has 2 important methods i.e.. apply() and compose()

For example,

```
Function<Integer, Double> half = a -> a / 2.0;
half = half.compose(a -> 3 * a);
System.out.println(half.apply(5));
```

c.Optional
Optional is used for handling NullPointerException.
Optional has different methods like
of(),ofNullable(),isPresent(),ifPresent(),orElse(),.orElseThrow()

For example,
```
package com.praveen.javautilitiesbypraveen.java8;
import java.util.Optional;
public class OptionalTest {
public static void main(String args[]) {
Integer[] numArray= new Integer[5];
// System.out.println(numArray[3].toString()); // It throws java.lang.NullPointerException
Optional <Integer>optional = Optional.ofNullable(numArray[3]);
if(optional.isPresent()) {
System.out.println(optional.get());
}
}

}
```

d.Consumer
Consumer takes single input and return nothing i.e. void.
For example,
```
Consumer<Integer> display = a -> System.out.println(a);
display.accept(10);
```

andThen() function also used for composed consumer

e.Supplier
Supplier takes no input and returns output.
For example,
```
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());
```

3.Double Colon :: Method Reference
Java8 provides new feature **method reference** to call a single method of functional interface.
Method reference can be used to refer both static and non-static(instance) methods.

```
package com.praveen.java8.lambda;

@FunctionalInterface
```

```
interface MethodReference {
void display();
}

public class StaticMethodReferenceDemo {

static void display() {
System.out.println("display");
}

public static void main(String args[]) {

/* With Methodreference */
MethodReference methodReference = StaticMethodReferenceDemo::display;
methodReference.display();

/* With Lambda */
MethodReference methodReferenceLambda = () ->
StaticMethodReferenceDemo.display();
methodReferenceLambda.display();
}

}
```

Rule1: Method names can be different.
Rule2: Parameter should be same in both methods.
Rule3: Method reference can be used in functional interface only.
Rule4: Return type of both methods can be different.

We can say MethodReference is alternative syntax for Lambda Expression.

```
package com.praveen.java8.lambda;

@FunctionalInterface
interface MethodReferenceNS {
void display();
}

public class NonStaticMethodReferenceDemo {

void display() {
System.out.println("display");
}
public static void main(String args[]) {

NonStaticMethodReferenceDemo obj = new NonStaticMethodReferenceDemo();

/* With Methodreference */
MethodReferenceNS methodReference = obj::display;
methodReference.display();

/* With Lambda */
MethodReferenceNS methodReferenceLambda = () -> obj.display();
methodReferenceLambda.display();
}

}
```

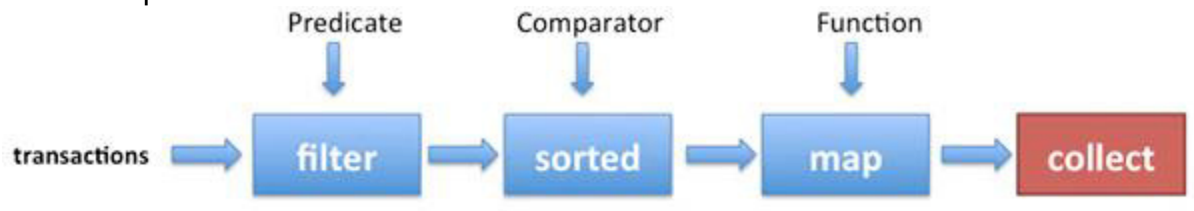
For non-static method reference we just need to create an object and call the method whereas for static method reference you can use Class Name directly.

4.Stream API

Stream API is used to process the collection of objects.

Why we need Stream?
1. We can achieve Functional Programming
2. Code Reduce
3. Bulk operation
4. Parallel streams make it very easy for multithreaded operations

Stream Pipeline
A Stream pipeline consist of source, a zero or more intermediate operations and a terminal operation.



Intermediate operations are
• filter(Predicate<T>)
• map(Function<T>)
• flatmap(Function<T>)
• sorted(Comparator<T>)
• peek(Consumer<T>)
• distinct()

• limit(long n)
• skip(long n)

Terminal operations are
forEach
toArray
reduce
collect
min
max
count
anyMatch
allMatch
noneMatch
findFirst
findAny

Stream filter()

filter allows you to filter the stream to process the stream. **Syntax** : Stream<T> filter(Predicate<? super T> predicate)

For example,

```
eList.stream()
.filter(e -> e.getEmpLocation().equalsIgnoreCase("Hyderabad"))
.forEach(System.out::println);
```

How filter works internally?

filter method takes a predicate as an input which is an functional interface. Java will convert the above filter syntax into below lines of code by adding a Predicate functional interface.
Java converts the above filter code by adding a Predicate class and overrides the test method as shown below:

```
eList.stream()
.filter(new Predicate<Employee>() {
public boolean test(Employee e) {
return "Hyderabad".equalsIgnoreCase(e.getEmpLocation());
}
})
.forEach(System.out::println);
```

usage of filter(),sort() and findFirst() methods

```
System.out.println(eList.stream()
.filter(e -> e.getEmpLocation().equalsIgnoreCase("Hyderabad"))
.sorted(new Comparator<Employee>(){
```

```
public int compare(Employee e1,Employee e2) {
return e1.getEmpName().compareToIgnoreCase(e2.getEmpName());
}
})
.findFirst().get());
```

map()

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

A stream consisting of the results of applying the given function to the elements of this stream. Map takes an input which describes how the value needs to be transformed into.
Suppose we want to get the age of the Employee whose name is Praveen

```
System.out.println(eList.stream()
.filter(e-> e.getEmpName().equalsIgnoreCase("praveen"))
.map(Employee::getEmpAge)
.findAny()
.orElse(0));
```

map() vs flatMap()
map() is used to transform one Stream into another by applying a function on each element and flatMap() does both transformation as well as flattening. The flatMap() function can take a Stream of List and return Stream of values combined from all those list.

```
listOfListOfNumber.add(Arrays.asList(2, 4));
listOfListOfNumber.add(Arrays.asList(3, 9));
listOfListOfNumber.add(Arrays.asList(4, 16));
System.out.println(listOfListOfNumber); // [[2, 4], [3, 9], [4, 16]]
System.out.println(listOfListOfNumber.stream()
.flatMap( list -> list.stream())
.collect(Collectors.toList())); // [2, 4, 3, 9, 4, 16]
```

Collectors

1.**joining()**
Collectors.joining will join all the results with delimiter specified in the parameter.

```
System.out.println(eList.stream()
.map(Employee:: getEmpName)
.collect(Collectors.joining("|"))); // Praveen|Khaja|Varma|Hari|Krishna
```

2. **summaryStatistics()**
It is used to calculate the sum, min, max, avg and total count of the elements passed to this method

```
IntSummaryStatistics statistics= eList.stream()
.mapToInt(Employee::getEmpAge)
.summaryStatistics();
System.out.println(statistics.getCount());
System.out.println(statistics.getSum());
System.out.println(statistics.getMin());
System.out.println(statistics.getMax());
System.out.println(statistics.getAverage());
```

3.**partitioningBy()**
We can partition a set of stream in two based on certain condition. So it will create two streams – one which satisfies the condition and another which does not satisfies the conditions.
partitioningBy will return a map containing two keys
true – which holds the stream which satisfy the condition.
false - which holds the stream which does not satisfy the condition.

```
Map<Boolean,List<Employee>> partition=
eList.stream().collect(Collectors.partitioningBy(e ->
e.getEmpLocation().equals("Hyderabad")));
System.out.println("Employees working in Hyderabad Location "+partition.get(true)); //
[Employee [empName=Praveen, empId=149903, empAge=34,
empLocation=Hyderabad], Employee [empName=Hari, empId=89778, empAge=43,
empLocation=Hyderabad]]
System.out.println("Employees working in other Location "+partition.get(false)); //
[Employee [empName=Khaja, empId=250005, empAge=35, empLocation=Bangalore],
Employee [empName=Varma, empId=26767, empAge=36, empLocation=Singapore],
Employee [empName=Krishna, empId=22203, empAge=38, empLocation=SouthAfrica]]
```

4. **groupingBy()**
groupingBy() is used to group the stream based on the condition passed to the groupingBy()

```
Map<String,List<Employee>> groupBy = eList.stream()
.collect(Collectors.groupingBy(Employee::getEmpLocation));
System.out.println(groupBy); // {Singapore=[Employee [empName=Varma,
empId=26767, empAge=36, empLocation=Singapore]], SouthAfrica=[Employee
[empName=Krishna, empId=22203, empAge=38, empLocation=SouthAfrica]],
Hyderabad=[Employee [empName=Praveen, empId=149903, empAge=34,
empLocation=Hyderabad], Employee [empName=Hari, empId=89778, empAge=43,
empLocation=Hyderabad]], Bangalore=[Employee [empName=Khaja, empId=250005,
empAge=35, empLocation=Bangalore]]}
```

5. **mappingBy**
mappingBy() allows us to pick the particular property of the Object to store into map rather than storing the complete Object
Map<String,Set<String>> mappingBy = eList.stream()
.collect(Collectors.groupingBy(Employee::getEmpLocation,
Collectors.mapping(Employee::getEmpName, Collectors.toSet())));
System.out.println(mappingBy); // {Singapore=[Varma], SouthAfrica=[Krishna], Hyderabad=[Hari, Praveen], Bangalore=[Khaja]}

5. New Date-Time API

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :

1. Not thread safe : Unlike old java.util.Date which is not thread safe the new date-time API is immutable and doesn't have setter methods.

2. Less operations : In old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

1. Local : Simplified date-time API with no complexity of timezone handling.
2. Zoned : Specialized date-time API to deal with various timezones.

1. LocalDate/LocalTime: LocalDate and LocalTime Classes is introduce where timezones are not required.

2. Zone DateTime API: It is used when time zone is to be

3.ChronoUnit Enum: ChronoUnit enum is added in the new Java 8 API which is Used to represent day, month, etc and it is available in java.time.temporal package.

For Example,

```
package com.praveen.java8.datetimeapi;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.Period;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoUnit;
```

```
public class NewDateTimeApiExamples {

public static void main(String args[]) {

// current date and time
LocalDateTime currentTime = LocalDateTime.now();
System.out.println(“Current DateTime: ” + currentTime);

LocalDate date1 = currentTime.toLocalDate();
System.out.println(“date1: ” + date1);

Month month = currentTime.getMonth();
int day = currentTime.getDayOfMonth();
int seconds = currentTime.getSecond();

System.out.println(“Month: ” + month + ” day: ” + day + ” seconds: ” + seconds);

// current date and time
ZonedDateTime date = ZonedDateTime.parse(“2019-07-
28T10:14:20+06:30[Asia/Kolkata]”);
System.out.println(“date: ” + date);

// Get the current date
LocalDate currentDate = LocalDate.now();
System.out.println(“Current date: ” + currentDate);

// add 1 week to the current date
LocalDate nextWeek = currentDate.plus(1, ChronoUnit.WEEKS);
System.out.println(“Next week: ” + nextWeek);

// add 2 month to the current date
LocalDate nextMonth = currentDate.plus(2, ChronoUnit.MONTHS);
System.out.println(“Next month: ” + nextMonth);

// add 3 year to the current date
LocalDate nextYear = currentDate.plus(3, ChronoUnit.YEARS);
System.out.println(“Next year: ” + nextYear);

// add 10 years to the current date
LocalDate nextDecade = currentDate.plus(1, ChronoUnit.DECADES);
System.out.println(“Next ten year: ” + nextDecade);

// comparing dates
LocalDate date2 = LocalDate.of(2014, 1, 15);
LocalDate date3 = LocalDate.of(2019, 7, 28);

if (date2.isAfter(date3)) {
System.out.println(“date2 comes after date3”);
} else {
System.out.println(“date2 comes before date3”);
}

// check Leap year
if (date1.isLeapYear()) {
System.out.println(“This year is Leap year”);
} else {
System.out.println(date1.getYear() + ” is not a Leap year”);
}

//How many days, month between two dates
LocalDate newDate = LocalDate.of(2019, Month.DECEMBER, 14);
Period periodTonewDate = Period.between(date1, newDate);
System.out.println(“Months left between today and newDate : ” +
periodTonewDate.getMonths() );

}

}
```

Output

```
Current DateTime: 2019-07-28T02:26:31.367

date1: 2019-07-28

Month: JULY

day: 28

seconds: 31

date: 2019-07-28T10:14:20+05:30[Asia/Kolkata]

Current date: 2019-07-28

Next week: 2019-08-04

Next month: 2019-09-28
Next year: 2022-07-28

Next ten year: 2029-07-28

date2 comes before date3

2019 is not a Leap yearMonths left between today and newDate : 4
```