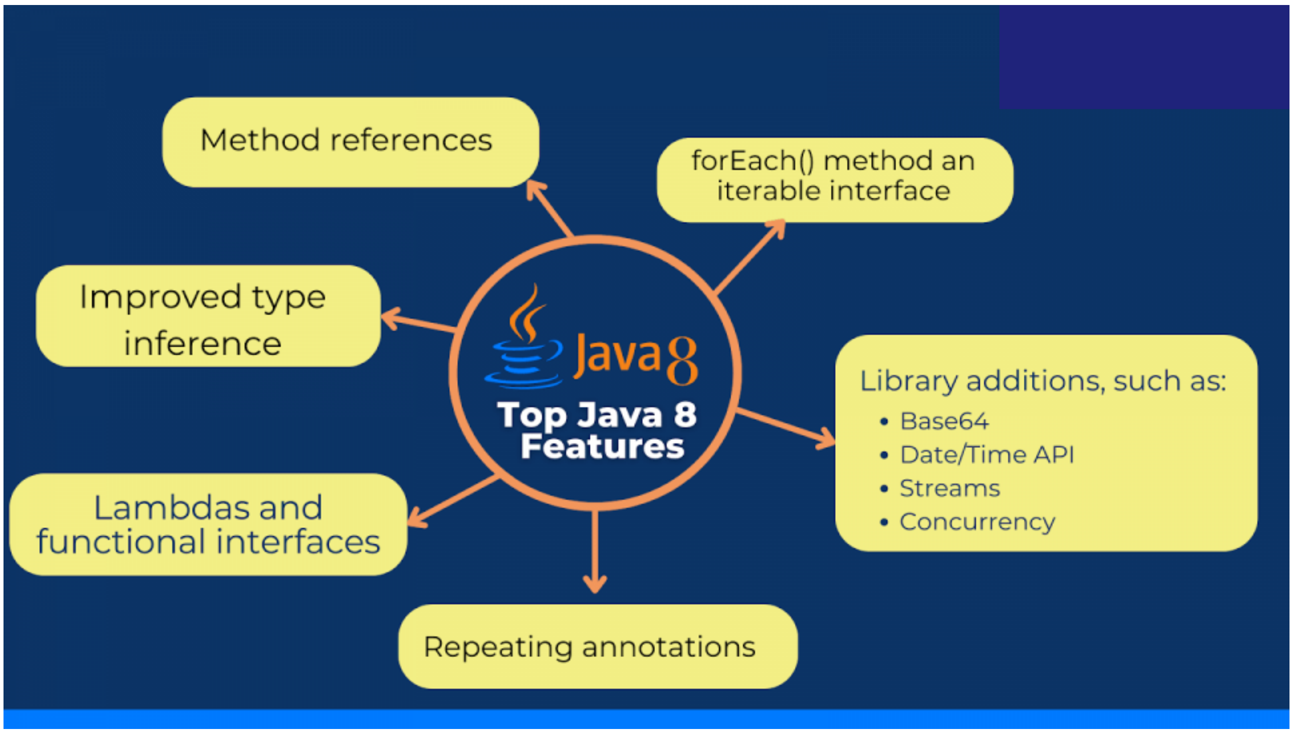
Java 8 - Overview

JAVA 8 is a major feature release of JAVA programming language development. Its initial version was released on 18 March 2014. With the Java 8 release, Java provided supports for functional programming, new JavaScript engine, new APIs for date time manipulation, new streaming API, etc.



New Features

- **Lambda expression** – Adds functional processing capability to Java.
- **Method references** – Referencing functions by their names instead of invoking them directly. Using functions as parameter.
- **Default method** – Interface to have default method implementation.
- **New tools** – New compiler tools and utilities are added like ‘jdeps’ to figure out dependencies.
- **Stream API** – New stream API to facilitate pipeline processing.
- **Date Time API** – Improved date time API.

- **Optional** – Emphasis on best practices to handle null values properly.

Java 8 Functional Interface

Functional Interface is an interface with only single abstract method. As a functional interface can have only one abstract method that's why it is also known as Single Abstract Method Interfaces or SAM Interfaces. We can either create our own functional interface or can use predefined functional interfaces provided by java.

Rules to define a functional interface:

A functional interface can have only one abstract method. Along with the one abstract method, it can have any number of default and static methods. It can also have methods of object class.

```
@FunctionalInterface
interface AddInterface{
    void add(int a, int b);
}

public class FunctionalInterfaceExample implements
AddInterface {
    public void add(int a, int b){
        System.out.println(a+b);
     }
    public static void main(String args[]){
        FunctionalInterfaceExample fie = new
FunctionalInterfaceExample();
        fie.add(10, 20);
    }
}
```

A functional interface can extends another interface only when it does not have any abstract method.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Java8Tester {

   public static void main(String args[]) {
      List<Integer> list = Arrays.asList(1, 2, 3,
4, 5, 6, 7, 8, 9);

      // Predicate<Integer> predicate = n -> true
      // n is passed as parameter to test method
of Predicate interface
      // test method will always return true no
matter what value n has.

      System.out.println("Print all numbers:");

      //pass n as parameter
      eval(list, n->true);

      // Predicate<Integer> predicate1 = n -> n%2
== 0
      // n is passed as parameter to test method
of Predicate interface
      // test method will return true if n%2 comes
to be zero

      System.out.println("Print even numbers:");
      eval(list, n-> n%2 == 0 );

      // Predicate<Integer> predicate2 = n -> n >
3
      // n is passed as parameter to test method
of Predicate interface
      // test method will return true if n is
greater than 3.
```

```
      System.out.println("Print numbers greater
than 3:");
      eval(list, n-> n > 3 );
   }

   public static void eval(List<Integer> list,
Predicate<Integer> predicate) {

      for(Integer n: list) {

         if(predicate.test(n)) {
            System.out.println(n + " ");
         }
      }
   }
}
```

Java 8 Lambda Expression

As java is an object oriented language and everything in java are objects, except primitive types. We need object references to call any method. But what in case of Java script. We can define functions anywhere we want, can assign functions to any variables, and can pass functions as an input parameters to a method. It is called functional programming language. Lambda expression is a way to visualize this functional programming in the java object oriented world. It enables to pass a functionality as a method argument. In java, lambda expressions are similar to functional programming, but not 100%.

Lambda expression is used to provide the implementation of functional interface. A Functional Interface is an interface with only single abstract method. In case of lambda expression, we don’t need to define the method again for providing the implementation so it saves lots of coding efforts.

Java Lambda Expression Syntax

(argument-list) -> {function-body}

Where:
Argument-list: It can be empty or non-empty as well.
Arrow notation/lambda notation: It is used to link arguments-list and body of expression.
Function-body: It contains expressions and statements for lambda expression.

Lambda expression advantage:

- It reduce lines of code.
- It provides sequential and parallel execution support.
- It provides a way to pass behaviors into methods.

Simple example without lambda expression

```
@FunctionalInterface
interface AddInterface{
    void add(int a, int b);
}

public class FunctionalInterfaceExample {
    public static void main(String args[]){
        //without lambda, AddInterface
implementation using anonymous class
        AddInterface addInterface=new
AddInterface(){
            public void add(int a, int b) {
             System.out.println(a + b);
            }
        };
        addInterface.add(10, 20);
    }
}
```

Simple example with lambda expression

```
@FunctionalInterface
interface AddInterface{
    void add(int a, int b);
}

public class LambdaExpressionExample {
    public static void main(String args[]){
        //Using lambda expressions
        AddInterface addInterface=(a, b)->{
             System.out.println(a + b);
        };
        addInterface.add(10, 20);
    }
}
```

```
public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester tester = new Java8Tester();

      //with type declaration
      MathOperation addition = (int a, int b) -> a
+ b;

      //with out type declaration
      MathOperation subtraction = (a, b) -> a - b;

      //with return statement along with curly
braces
      MathOperation multiplication = (int a, int
b) -> { return a * b; };

      //without return statement and without curly
braces
      MathOperation division = (int a, int b) -> a
/ b;
```

```
   }

      System.out.println("10 + 5 = " +
tester.operate(10, 5, addition));
      System.out.println("10 - 5 = " +
tester.operate(10, 5, subtraction));
      System.out.println("10 x 5 = " +
tester.operate(10, 5, multiplication));
      System.out.println("10 / 5 = " +
tester.operate(10, 5, division));

      //without parenthesis
      GreetingService greetService1 = message ->
      System.out.println("Hello " + message);

      //with parenthesis
      GreetingService greetService2 = (message) ->
      System.out.println("Hello " + message);

      greetService1.sayMessage("Mahesh");
      greetService2.sayMessage("Suresh");
   }

   interface MathOperation {
      int operation(int a, int b);
   }

   interface GreetingService {
      void sayMessage(String message);
   }

   private int operate(int a, int b, MathOperation
mathOperation) {
      return mathOperation.operation(a, b);
   }
}
```

Java 8 - Method References

Method references help to point to methods by their names. A method reference is described using "::" symbol. A method reference can be used to point the following types of methods –

- Static methods
- Instance methods
- Constructors using new operator (TreeSet::new)

Method Reference Example

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

public class Java8Tester {

   public static void main(String args[]) {
      List<String> names = new 
ArrayList<String>();

      names.add("Mahesh");
      names.add("Suresh");
      names.add("Ramesh");
      names.add("Naresh");
      names.add("Kalpesh");

      names.forEach(System.out::println);
   }
}
```

Why default method?

For example we have an interface which has multiple methods, and multiple classes are implementing this interface. One of the method implementation can be common across the class. We can make that method as a default method, so that the implementation is common for all classes.

Second scenario where we have already existing application, for a new requirements we have to add a method to the existing interface. If we add new method then we need to implement it throughout the implementation classes. By using the Java 8 default method we can add a default implementation of that method which resolves the problem.

Java default methods

Java default methods are defined inside the interface and tagged with default keyword. Default methods are non-abstract methods. We can override default methods to provide more specific implementation.

```java
interface Display{
    //Default method
    default void display(){
        System.out.println("Hello Jai");
    }
    //Abstract method
    void displayMore(String msg);
}
public class DefaultMethods implements Display{
    //implementing abstract method
    public void displayMore(String msg){
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        //calling default method
        dm.display();
        //calling abstract method
        dm.displayMore("Hello Java8");
    }
}
```

```
Output:

Hello Jai
Hello Java8
```

Static methods in interfaces are similar to the default methods. The only difference is that we cannot override these methods in the classes that implements these interfaces.

```
interface Display{
    //Default method
    default void display(){
        System.out.println("Hello Jai");
    }
    //Abstract method
    void displayMore(String msg);
    //Static method
    static void show(String msg){
        System.out.println(msg);
    }
}
public class StaticMethodTest implements Display{
    //implementing abstract method
    public void displayMore(String msg){
        System.out.println(msg);
    }
    public static void main(String[] args) {
       StaticMethodTest smt = new
StaticMethodTest();
        //calling default method
        smt.display();
        //calling abstract method
        smt.displayMore("Hello Static Methods");
        //calling static method
        Display.show("Hello Java");
    }
}
```

}

Output:

```
Hello Jai
Hello Static Methods
Hello Java
```

Another Example

```java
public class Java8Tester {

   public static void main(String args[]) {
      Vehicle vehicle = new Car();
      vehicle.print();
   }
}

interface Vehicle {

   default void print() {
      System.out.println("I am a vehicle!");
   }

   static void blowHorn() {
      System.out.println("Blowing horn!!!");
   }
}

interface FourWheeler {

   default void print() {
      System.out.println("I am a four wheeler!");
   }
}

class Car implements Vehicle, FourWheeler {

   public void print() {
      Vehicle.super.print();
      FourWheeler.super.print();
      Vehicle.blowHorn();
      System.out.println("I am a car!");
   }
}
```

Output:

```
I am a vehicle!
I am a four wheeler!
Blowing horn!!!
I am a car!
```

Java 8 Stream ForEach Method Example

The java.util.stream is a sequence of elements supporting sequential and parallel aggregate operations.
The Stream.forEach() method works same as of for loop. It iterates through each element and performs the specified operations.

Java 8 stream forEach method example

```
import java.util.Arrays;
import java.util.List;

public class Test{
    public static void main(String[] args) {
       List<String> names = Arrays.asList("jai",
"hemant", "mahesh", "vishal", "vivek");

names.stream().forEach(System.out::println);
    }
}
```

Output:

```
jai
hemant
mahesh
vishal
vivek
```

What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream –

- **Sequence of elements** – A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
- **Source** – Stream takes Collections, Arrays, or I/O resources as input source.
- **Aggregate operations** – Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
- **Pipelining** – Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
- **Automatic iterations** – Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Generating Streams

With Java 8, Collection interface has two methods to generate a Stream.

- **stream()** – Returns a sequential stream considering collection as its source.
- **parallelStream()** – Returns a parallel Stream considering collection as its source.

```
List<String> strings = Arrays.asList("abc", "",
"bc", "efg", "abcd","", "jkl");
List<String> filtered =
strings.stream().filter(string -> !
string.isEmpty()).collect(Collectors.toList());
```

forEach

Stream has provided a new method ‘forEach’ to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::printl
n);
```

map

The ‘map’ method is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3,
7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
numbers.stream().map( i ->
i*i).distinct().collect(Collectors.toList());
```

filter

The ‘filter’ method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

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

//get count of empty string
int count = strings.stream().filter(string ->
string.isEmpty()).count();
```

limit

The ‘limit’ method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::printl
n);
```

sorted

The ‘sorted’ method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

```
Random random = new Random();
random.ints().limit(10).sorted().forEach(System.ou
t::println);
```

Parallel Processing

parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

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

//get count of empty string
long count =
strings.parallelStream().filter(string ->
string.isEmpty()).count();
```

It is very easy to switch between sequential and parallel streams.

Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

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

```
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);
```

Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

```
List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats =
numbers.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());

import java.util.ArrayList;
import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;
import java.util.Map;
```

```
public class Java8Tester {

   public static void main(String args[]) {
      System.out.println("Using Java 7: ");

      // Count empty strings
      List<String> strings = Arrays.asList("abc",
"", "bc", "efg", "abcd","", "jkl");
      System.out.println("List: " +strings);
      long count =
getCountEmptyStringUsingJava7(strings);

      System.out.println("Empty Strings: " +
count);
      count = getCountLength3UsingJava7(strings);

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

      //Eliminate empty string
      List<String> filtered =
deleteEmptyStringsUsingJava7(strings);
      System.out.println("Filtered List: " +
filtered);

      //Eliminate empty string and join using
comma.
      String mergedString =
getMergedStringUsingJava7(strings,", ");
      System.out.println("Merged String: " +
mergedString);
      List<Integer> numbers = Arrays.asList(3, 2,
2, 3, 7, 3, 5);

      //get list of square of distinct numbers
      List<Integer> squaresList =
getSquares(numbers);
```

```
      System.out.println("Squares List: " +
squaresList);
      List<Integer> integers =
Arrays.asList(1,2,13,4,15,6,17,8,19);

      System.out.println("List: " +integers);
      System.out.println("Highest number in List :
" + getMax(integers));
      System.out.println("Lowest number in List :
" + getMin(integers));
      System.out.println("Sum of all numbers : " +
getSum(integers));
      System.out.println("Average of all numbers :
" + getAverage(integers));
      System.out.println("Random Numbers: ");

      //print ten random numbers
      Random random = new Random();

      for(int i = 0; i < 10; i++) {
         System.out.println(random.nextInt());
      }

      System.out.println("Using Java 8: ");
      System.out.println("List: " +strings);

      count = strings.stream().filter(string-
>string.isEmpty()).count();
      System.out.println("Empty Strings: " +
count);

      count = strings.stream().filter(string ->
string.length() == 3).count();
      System.out.println("Strings of length 3: " +
count);

      filtered = strings.stream().filter(string
->!string.isEmpty()).collect(Collectors.toList());
```

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

      mergedString =
strings.stream().filter(string ->!
string.isEmpty()).collect(Collectors.joining(",
"));
      System.out.println("Merged String: " +
mergedString);

      squaresList = numbers.stream().map( i
->i*i).distinct().collect(Collectors.toList());
      System.out.println("Squares List: " +
squaresList);
      System.out.println("List: " +integers);

      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.ints().limit(10).sorted().forEach(System.ou
t::println);

      //parallel processing
      count =
strings.parallelStream().filter(string ->
string.isEmpty()).count();
```

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

   private static int
getCountEmptyStringUsingJava7(List<String>
strings) {
      int count = 0;

      for(String string: strings) {

         if(string.isEmpty()) {
            count++;
         }
      }
      return count;
   }

   private static int
getCountLength3UsingJava7(List<String> strings) {
      int count = 0;

      for(String string: strings) {

         if(string.length() == 3) {
            count++;
         }
      }
      return count;
   }

   private static List<String>
deleteEmptyStringsUsingJava7(List<String> strings)
{
      List<String> filteredList = new
ArrayList<String>();

      for(String string: strings) {
         if(!string.isEmpty()) {
             filteredList.add(string);
         }
      }
      return filteredList;
   }

   private static String
getMergedStringUsingJava7(List<String> strings,
String separator) {
      StringBuilder stringBuilder = new
StringBuilder();

      for(String string: strings) {

         if(!string.isEmpty()) {
            stringBuilder.append(string);
            stringBuilder.append(separator);
         }
      }
      String mergedString =
stringBuilder.toString();
      return mergedString.substring(0,
mergedString.length()-2);
   }

   private static List<Integer>
getSquares(List<Integer> numbers) {
      List<Integer> squaresList = new
ArrayList<Integer>();

      for(Integer number: numbers) {
         Integer square = new
Integer(number.intValue() * number.intValue());

         if(!squaresList.contains(square)) {
            squaresList.add(square);
            }
        }
        return squaresList;
    }

    private static int getMax(List<Integer>
numbers) {
        int max = numbers.get(0);

        for(int i = 1;i < numbers.size();i++) {

            Integer number = numbers.get(i);

            if(number.intValue() > max) {
                max = number.intValue();
            }
        }
        return max;
    }

    private static int getMin(List<Integer>
numbers) {
        int min = numbers.get(0);

        for(int i= 1;i < numbers.size();i++) {
            Integer number = numbers.get(i);

            if(number.intValue() < min) {
                min = number.intValue();
            }
        }
        return min;
    }

    private static int getSum(List numbers) {
        int sum = (int)(numbers.get(0));

        for(int i = 1;i < numbers.size();i++) {
         sum += (int)numbers.get(i);
      }
      return sum;
   }

   private static int getAverage(List<Integer>
numbers) {
      return getSum(numbers) / numbers.size();
   }
}
```

Output:

```
Using Java 7:
List: [abc, , bc, efg, abcd, , jkl]
Empty Strings: 2
Strings of length 3: 3
Filtered List: [abc, bc, efg, abcd, jkl]
Merged String: abc, bc, efg, abcd, jkl
Squares List: [9, 4, 49, 25]
List: [1, 2, 13, 4, 15, 6, 17, 8, 19]
Highest number in List : 19
Lowest number in List : 1
Sum of all numbers : 85
Average of all numbers : 9
Random Numbers:
-1279735475
903418352
-1133928044
-1571118911
628530462
18407523
-881538250
-718932165
270259229
421676854
Using Java 8:
List: [abc, , bc, efg, abcd, , jkl]
```

```
Empty Strings: 2
Strings of length 3: 3
Filtered List: [abc, bc, efg, abcd, jkl]
Merged String: abc, bc, efg, abcd, jkl
Squares List: [9, 4, 49, 25]
List: [1, 2, 13, 4, 15, 6, 17, 8, 19]
Highest number in List : 19
Lowest number in List : 1
Sum of all numbers : 85
Average of all numbers : 9.444444444444445
Random Numbers:
-1009474951
-551240647
-2484714
181614550
933444268
1227850416
1579250773
1627454872
1683033687
1798939493
Empty Strings: 2
```

Java 8 - Optional Class

Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values

Class Declaration

Following is the declaration for **java.util.Optional<T>** class –

```
public final class Optional<T> extends Object

import java.util.Optional;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8Tester = new Java8Tester();
      Integer value1 = null;
      Integer value2 = new Integer(10);

      //Optional.ofNullable - allows passed
parameter to be null.
      Optional<Integer> a =
Optional.ofNullable(value1);

      //Optional.of - throws NullPointerException
if passed parameter is null
      Optional<Integer> b = Optional.of(value2);
      System.out.println(java8Tester.sum(a,b));
   }

   public Integer sum(Optional<Integer> a,
Optional<Integer> b) {
      //Optional.isPresent - checks the value is
present or not

      System.out.println("First parameter is
present: " + a.isPresent());
      System.out.println("Second parameter is
present: " + b.isPresent());

      //Optional.orElse - returns the value if
present otherwise returns
      //the default value passed.
      Integer value1 = a.orElse(new Integer(0));

      //Optional.get - gets the value, value
should be present
      Integer value2 = b.get();
      return value1 + value2;
   }
}
```

Output:

```
First parameter is present: false
Second parameter is present: true
10
```

Java 8 - New Date/Time API

Java 8 introduces a new date-time API under the package java.time. Following are some of the important classes introduced in java.time package.

- **Local** – Simplified date-time API with no complexity of timezone handling.
- **Zoned** – Specialized date-time API to deal with various timezones.

Local Date-Time API

LocalDate/LocalTime and LocalDateTime classes simplify the development where timezones are not required. Let's see them in action.

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testLocalDateTime();
   }

   public void testLocalDateTime() {
      // Get the 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);

      LocalDateTime date2 = 
currentTime.withDayOfMonth(10).withYear(2012);
      System.out.println("date2: " + date2);

      //12 december 2014
      LocalDate date3 = LocalDate.of(2014, 
Month.DECEMBER, 12);
      System.out.println("date3: " + date3);

      //22 hour 15 minutes
      LocalTime date4 = LocalTime.of(22, 15);
      System.out.println("date4: " + date4);

      //parse a string
      LocalTime date5 = 
LocalTime.parse("20:15:30");
      System.out.println("date5: " + date5);
   }
}
```

Output:

```
Current DateTime: 2014-12-09T11:00:45.457
date1: 2014-12-09
Month: DECEMBERday: 9seconds: 45
date2: 2012-12-10T11:00:45.457
date3: 2014-12-12
date4: 22:15
```

```
date5: 20:15:30
```

Zoned Date-Time API

Zoned date-time API is to be used when time zone is to be considered.

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testZonedDateTime();
   }

   public void testZonedDateTime() {
      // Get the current date and time
      ZonedDateTime date1 = 
ZonedDateTime.parse("2007-12-03T10:15:30+05:30[Asi
a/Karachi]");
      System.out.println("date1: " + date1);

      ZoneId id = ZoneId.of("Europe/Paris");
      System.out.println("ZoneId: " + id);

      ZoneId currentZone = ZoneId.systemDefault();
      System.out.println("CurrentZone: " + 
currentZone);
   }
}

Output:

date1: 2007-12-03T10:15:30+05:00[Asia/Karachi]
ZoneId: Europe/Paris
CurrentZone: Etc/UTC
```

Chrono Units Enum

java.time.temporal.ChronoUnit enum is added in Java 8 to replace the integer values used in old API to represent day, month, etc. Let us see them in action.

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testChromoUnits();
   }

   public void testChromoUnits() {
      //Get the current date
      LocalDate today = LocalDate.now();
      System.out.println("Current date: " +
today);

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

      //add 1 month to the current date
      LocalDate nextMonth = today.plus(1,
ChronoUnit.MONTHS);
      System.out.println("Next month: " +
nextMonth);

      //add 1 year to the current date
      LocalDate nextYear = today.plus(1,
ChronoUnit.YEARS);
      System.out.println("Next year: " +
nextYear);
```

```
      //add 10 years to the current date
      LocalDate nextDecade = today.plus(1,
ChronoUnit.DECADES);
      System.out.println("Date after ten year: " +
nextDecade);
   }
}

Output:

Current date: 2014-12-10
Next week: 2014-12-17
Next month: 2015-01-10
Next year: 2015-12-10
Date after ten year: 2024-12-10
```

Period and Duration

With Java 8, two specialized classes are introduced to deal with the time differences.

- **Period** – It deals with date based amount of time.
- **Duration** – It deals with time based amount of time.

```
import java.time.temporal.ChronoUnit;

import java.time.LocalDate;
import java.time.LocalTime;
import java.time.Duration;
import java.time.Period;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testPeriod();
      java8tester.testDuration();
   }
   public void testPeriod() {
      //Get the current date
      LocalDate date1 = LocalDate.now();
      System.out.println("Current date: " +
date1);

      //add 1 month to the current date
      LocalDate date2 = date1.plus(1,
ChronoUnit.MONTHS);
      System.out.println("Next month: " + date2);

      Period period = Period.between(date2,
date1);
      System.out.println("Period: " + period);
   }

   public void testDuration() {
      LocalTime time1 = LocalTime.now();
      Duration twoHours = Duration.ofHours(2);

      LocalTime time2 = time1.plus(twoHours);
      Duration duration = Duration.between(time1,
time2);

      System.out.println("Duration: " + duration);
   }
}
```

Output:

```
Current date: 2014-12-10
Next month: 2015-01-10
Period: P-1M
Duration: PT2H
```

Backward Compatibility

A toInstant() method is added to the original Date and Calendar objects, which can be used to convert them to the new Date-Time API. Use an ofInstant(Insant,ZoneId) method to get a LocalDateTime or ZonedDateTime object. Let us see them in action.

```
import java.time.LocalDateTime;
import java.time.ZonedDateTime;

import java.util.Date;

import java.time.Instant;
import java.time.ZoneId;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testBackwardCompatability();
   }

   public void testBackwardCompatability() {
      //Get the current date
      Date currentDate = new Date();
      System.out.println("Current date: " + 
currentDate);

      //Get the instant of current date in terms 
of milliseconds
      Instant now = currentDate.toInstant();
      ZoneId currentZone = ZoneId.systemDefault();

      LocalDateTime localDateTime = 
LocalDateTime.ofInstant(now, currentZone);
      System.out.println("Local date: " + 
localDateTime);
```

```
      ZonedDateTime zonedDateTime =
ZonedDateTime.ofInstant(now, currentZone);
      System.out.println("Zoned date: " +
zonedDateTime);
   }
}
```

Output:

```
Current date: Wed Dec 10 05:44:06 UTC 2014
Local date: 2014-12-10T05:44:06.635
Zoned date: 2014-12-10T05:44:06.635Z[Etc/UTC]
```

Java 8 - Base64

With Java 8, Base64 has finally got its due. Java 8 now has inbuilt encoder and decoder for Base64 encoding. In Java 8, we can use three types of Base64 encoding.

- **Simple** – Output is mapped to a set of characters lying in A-Za-z0-9+/. The encoder does not add any line feed in output, and the decoder rejects any character other than A-Za-z0-9+/.
- **URL** – Output is mapped to set of characters lying in A-Za-z0-9+_. Output is URL and filename safe.
- **MIME** – Output is mapped to MIME friendly format. Output is represented in lines of no more than 76 characters each, and uses a carriage return '\r' followed by a linefeed '\n' as the line separator. No line separator is present to the end of the encoded output.