### Advantages of Java 8?

* Compact, readable, and reusable code.
* Less boilerplate code.
* Parallel operations and execution.

### **What new features did Java 8 introduce?**

* A Date and time API
* A new language called Lambda Expressions that treats actions as objects
* Method References, which enable defining Lambda Expressions by referring to methods directly using their names
* Default methods, which give users the ability to add full implementations in interfaces besides abstract methods
* Stream API, a special iterator class that allows processing object collections in a functional manner

**Functional Programming ?**

Functional programming is a programming paradigm where the focus is on “what to solve” in contrast to a procedural paradigm where the main focus is “how to solve.” **In other words, It is a declarative style of programming rather than imperative.** The core of Functional Programming is to solve computational problems as evaluation of **mathematical functions.**

**In functional programming we can use function as expression by lambda expression.**

Functional style language provides higher-order functions

**Higher-order functions:** In functional programming, functions are to be considered as first-class citizens. That is, so far in the legacy style of coding, we can do below stuff with objects.

* 1. We can pass **objects** to a function.
  2. We can create **objects** within function.
  3. We can return **objects** from a function.
  4. We can pass a **function** to a function.
  5. We can create a **function** within function.
  6. We can return a **function** from a function.

Higher order functions are that either takes one or more function as an argument or returns the function as a result.

Ex:

Private static Predicate<String> checkIfStartsWith (String name){

Return name -> name.startsWith(name);

}

In the above example, checkIfStartsWith() returns Function which takes 1 argument and returns the Predicate.

names.stream.filter(checkIfStartsWith(“Jo”)).collect(Collectors.toList());

Here above in statement in filter we are passing function and in return getting functional interface i.e predicate.

### **Anonymous Class**

Anonymous class is an inner class without a name, which means that we can declare and instantiate class at the same time. An anonymous class is used primarily when we want to use the class declaration once. Anonymous classes usually extend a subclass or implement an interface.

Ex:

ist.sort(comparator);

Comparator<Integer> comparator = new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o1 - o2;

}

};

Lambda Function

The expression which we can represent an Anonymous function is knows as Lambda function

Anonymous means Nameless /Unknown

Anonymous function : A method which don’t have any name or modifier

Syntax:

Parameter expression body

() -> { syso(“Lambda Expression”); };

### syntax of a lambda expression?

FunctionalInterface fi = (String name) -> {

System.out.println("Hello "+name);

**return** "Hello "+name;

}

1. List of Arguments/Params:

(String name)

A list of params is passed in () round brackets. It can have zero or more params. Declaring the type of parameter is optional and can be inferred for the context.

2. Arrow Token:

->   
Arrow token is known as the lambda arrow operator. It is used to separate the parameters from the body, or it points the list of arguments to the body. 3. Expression/Body:

{

System.out.println("Hello "+name);

**return** "Hello "+name;

}

A body can have expressions or statements. {} curly braces are only required when there is more than one line. In one statement, the return type is the same as the return type of the statement. In other cases, the return type is either inferred by the return keyword or void if nothing is returned.

**Common ways to use the lambda expression**

Assignment to a functional Interface —> Predicate<String> stringPredicate = s -> s.isEmpty();  
Can be passed as a parameter that has a functional type —> stream.filter(s -> s.isEmpty())  
Returning it from a function —> return s -> s.isEmpty()  
Casting it to a functional type —> (Predicate<String>) s -> s.isEmpty()

**Method Reference**

* Method Reference in java 8 makes the code simple and more readable than lambda expression.
* Method reference refers to the method via the use of an :: operator.
* A method reference in Java 8 can execute only a single method call like a lambda expression but a shorter code.

Ex:

str -> System.out.println(str)

By method reference, it will change to this:

System.out::println

In method reference, the :: operator divides the method name from the class or object name.

In the statement System.out::println, System is a class defined in the java.lang package, out is an instance of the type PrintStream, which is a public and static member field of the System class, and println() is a public method of all instances of the PrintStream class.

Functional Interface

The interface which contains only one abstract method but can have multiple default and static methods is known as functional interface.

For declaring Functional Interfaces @FunctionalInterface annotation is optional to use. If this annotation is used for interfaces with more than one abstract method, it will generate a compiler error.

Ex: Runnable -> run()

Comparable -> compareTo()

Comparator -> compare()

Ex for Lambda Expression

We don’t need {} if one statement is there

@FunctionalInterface  
interface ToTest{  
 void test();  
}  
  
public class Main {  
 public static void main(String[] args) {  
   
 ToTest toTest= () -> System.*out*.println("Hello");  
   
 toTest.test();  
 }

}

Stream

Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result. (i.e stream is a sequence of objects)

Ex:

MyClass{

Void print(int num){ syso(num);}

    Psvm{

List<Integer> number = Arrays.asList(2,3,4,5);

    List<Integer> square = number.stream().forEach(MyClass::print);

//Method Reference syntax Class::method

//Alternate way

List<Integer> square = number.stream().forEach(System.out::println);

}

}

Components of the stream are:

* A data source (Array,List, Stream,etc)
* Set of Intermediate Operations to process the data source
* Single Terminal Operation that produces the result

Filter, map

reduce()

List<Integer> integerList= Arrays.*asList*(1,2,3);  
System.*out*.println(integerList.stream().reduce(0,(a, b)->a+b));

Will reduce to single value 1st parameter is initial value

Here in background this will happen

reduce(int aggregate,int nextNumber){

aggregate=aggregate+nextNumber;//0+1 //1+2 //3+3 =6

}

//In Integer class already sum method will be present so use below

System.*out*.println(integerList.stream().reduce(0,Integer::sum));

The distinct() method returns a stream of unique elements. It is a stateful intermediate operation and returns a new stream. ...

The sorted() method returns a stream consisting of elements in the sorted natural order. We can also sort into reverse order as well

Count() methods returns ccount of numbers

**peek()**

**This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline.** returns a new Stream consisting of all the elements from the original Stream without changing the result.

**Ex:** List<String> l1=Arrays.*asList*("one", "two",”three", "eleven","twelve");

l1.stream()  
 .peek(e -> System.*out*.println("Normal value: " + e))  
 .filter(e -> e.length() > 3)  
 .peek(e -> System.*out*.println("filtered value: " + e))  
 .map(String::toUpperCase)  
 .peek(e -> System.*out*.println("Mapped value: " + e))  
 .findFirst();  
  
*//output Normal value: one  
//Normal value: two  
//Normal value: three  
//filtered value: three  
//Mapped value: THREE*

You can see in above example that only upto Three is streamed and returned it even doesnot looked next elements. But in normal way, for loop we need to iterate all elements.So, performance is improved.

Streams are lazy because **intermediate operations are not evaluated until terminal operation is invoked**. Each intermediate operation creates a new stream, stores the provided operation/function and return the new stream. **All the intermediate operations are lazy like peek,map,filter. They are returning stream they only execute when terminal operation like findfirst or sum is executed.**

| **Sr. No.** | **Key** | **Intermediate Operations** |  | **Terminal Operations** |
| --- | --- | --- | --- | --- |
| 1 | Basic | These operations are used to pipeline other methods and to transform into the other streams |  | A terminal operation in Java is a method applied to a stream as the final step. |
| 2 | Return Type | They only return another stream. |  | They return final result. |
| 3 | Method | distinct(),sorted(),map,filter  These methods stream over elements and returns stream |  | Collect,forEach,reduce  These methods return single type |
| 4. | Use Case | These operations should be used to transform stream into another stream |  | They can be used to produce results. |

Intermediate Operations are of two kinds

Stateful : Elements need to be compared against one another (sort, distinct etc)

Stateless : No need for comparing with other elements (map, filter etc)

To complete some of the intermediate operations, some state is to be maintained, and such intermediate operations are called **stateful intermediate operations.** Parallel execution of these types of operations is complex.

For Eg: sorted() , distinct() , limit() , skip() etc.

Sending data elements to further steps in the pipeline stops till all the data is sorted for sorted() and stream data elements are stored in temporary data structures.

**Predicate**

A Predicate is a functional interface, which accepts an argument and returns a boolean. Usually, it is used to apply in a filter for a collection of objects.

A Predicate interface represents a boolean-valued-function of an argument. This is mainly used to filter data from a Java Stream. The filter method of a stream accepts a predicate to filter the data and return a new stream satisfying the predicate. A predicate has a

**boolean** test(T value); method which accepts an argument and returns a boolean value.

Ex: List<Integer> integerList= Arrays.*asList*(1,2,3,4);

//integerList.stream().filter(n->n%2==0).map(x->x\*x)

.forEach(x->System.*out*.println(x));

integerList.stream().filter(*integerPredicate*)  
.map(*integerFunction*).forEach(*consumer*);

public static Predicate<Integer> *integerPredicate*= new Predicate<Integer>() {  
 @Override  
 public boolean test(Integer num) {  
 return num%2==0;  
 }  
};

# Function

A **Function** is another in-build functional interface in java.util.function package, A Function interface is more of a generic one that takes one argument and produces a result. This has a Single Abstract Method (SAM) apply which accepts an argument of a type T and produces a result of type R. R apply(T var1);

One of the common use cases of this interface is Stream.map method.

Ex: Refer above example

integerList.stream().filter(*integerPredicate*)  
.map(*integerFunction*).forEach(*consumer*);

private static Function<Integer, Integer> *integerFunction*= new Function<Integer, Integer>() {  
 @Override  
 public Integer apply(Integer num) {  
 return num\*num;  
 }  
};

# Consumer

A Consumer is a functional interface that accepts a single input and returns no output. In layman’s language, as the name suggests the implementation of this interface consumes the input supplied to it. The accept method is the Single Abstract Method (SAM) which accepts a single argument of type T.

**void** accept(T t);

Ex:For printing elements we can use this consumer

integerList.stream().filter(*integerPredicate*)  
.map(*integerFunction*).forEach(*consumer*);

private static Consumer<Integer> *consumer*=new Consumer<Integer>() {  
 @Override  
 public void accept(Integer num) {  
 System.*out*.println(num);  
 }  
};

Supplier

It represents a function that does not take in any argument but produces a value of type T. It contains only one method.

T get();

Ex:

Arrays.*asList*(*citySupplier*.get()).forEach(System.*out*::println);

public static Supplier<String[]> *citySupplier* = new Supplier<String[]>() {  
 @Override  
 public String[] get() {  
 return new String[]{"Mumbai", "Delhi", "Goa", "Pune"};  
 }  
};

Supplier<Integer> supplier=()->{

Random random=new Random();

random.nextInt(100); //returns any number below hundred

};

So Lambda is like objects ex: when we x->x%2==0 is in background creating instance of predicate class.

BinaryOperator

It is **a functional interface and it extends BiFunction** . The BinaryOperator takes two arguments of the same type and returns a result of the same type of its arguments. The BiFunction takes two arguments of any type, and returns a result of any type

Integer apply(a,b);

Ex:

List<Integer> integerList= Arrays.*asList*(1,2,3,4);

System.*out*.println(integerList.stream().reduce(0,*IntegerBinaryOperator*));

private static BinaryOperator<Integer> *IntegerBinaryOperator*=new BinaryOperator<Integer>() {  
 @Override  
 public Integer apply(Integer a, Integer b) {  
 return a+b;  
 }  
};

Unary Operator

It takes a single operand that produces a result of the same type as its operand.T apply(T t)

Ex:

UnaryOperator<Integer> unaryOperator= (x)->3\*x;  
  
System.*out*.println(unaryOperator.apply(10));

Similar to Predicate,Function,Consumer there are BiPredicate, BiFunction and BiConsumer

Which accepts 2 parameters than single

Ex:

1 BiPredicate<Integer,String> biPredicate=(a, str) -> {  
 return a>5 && str.length()>5;  
 };  
  
 System.*out*.println(biPredicate.test(6,"Jhonny"));  
  
2 BiFunction<Integer,String,String> biFunction=(a, str) ->{  
 return a+" "+str;  
 };  
//3rd parameter is the return type

System.*out*.println(biFunction.apply(20,"Somu"));  
  
3 BiConsumer<Integer,String> biConsumer=(a,str)->{  
 System.*out*.println(a);  
 System.*out*.println(str);  
 };  
biConsumer.accept(10,"Ramu");

//functional interfaces for primitive types like int boxing and unboxing cause performance issues so we can use these below interfaces for primitive types.

IntBinaryOperator

IntConsumer

IntPredicate

IntFunction

IntSupplier

IntToDoubleFunction

IntToLongFunction

Behavior parameterization

The behavior parameterization enables separation of the behavior of the object, thereby enhancing the reusability of the code and creating a flexible API. Behavior parameterization lets you make your code more adaptive to changing requirements and saves on engineering efforts in the future.

Ex:

List<Integer> list1= *getCollect*(list, x -> x \* x);  
List<Integer> list2= *getCollect*(list, x -> x \* x \* x);

private static List<Integer> getCollect(List<Integer> list, Function<Integer, Integer> integerFunction) {  
 return list.stream().map(integerFunction).collect(Collectors.*toList*());  
}

Higher Order Functions

These are the functions that either takes one or more function as an argument or returns the function as a result.

Ex:

Private static Predicate<String> checkIfStartsWith (String name){

Return name -> name.startsWith(name);

}

In the above example, checkIfStartsWith() returns Function which takes 1 argument and returns the Predicate.

names.stream.filter(checkIfStartsWith(“Jo”)).collect(Collectors.toList());

In the above example, filter() is the higher-order function that takes Function as an argument and returns Predicate

//Refer program for custom class functional interface

Eventhough we stream over list it will be wrapper objects like Integer

Stream stream = Stream.*of*(1,2,3);

This also streams over wrapper integer

We can also stream over int primitive type by creating int array int[]

How to stream primitive int in dynamic way

*//Streaming through primitive int  
  
//will get sum from 1 to 9 without including 10* System.*out*.println(IntStream.*range*(1,10).sum());  
 *//sum from 1 to 10* System.*out*.println(IntStream.*rangeClosed*(1,10).sum());  
 *//peek allows to look into stream without making any changes to stream* System.*out*.println(IntStream.*iterate*(1,e->e+2).limit(10).peek(System.*out*::println).sum());  
 *//even numbers sum from 2 to 20* System.*out*.println(IntStream.*iterate*(2,e->e+2).limit(10).peek(System.*out*::println).sum());  
 *//sum of first 10 powers of 2* System.*out*.println(IntStream.*iterate*(2,e->e\*2).limit(10).peek(System.*out*::println).sum());  
 *//to convert into list by boxed(i.e from int to Integer) and print* System.*out*.println(IntStream.*iterate*(2,e->e\*2).limit(10).boxed().collect(Collectors.*toList*()));  
  
 *//Factorial of 20* System.*out*.println(LongStream.*rangeClosed*(1,20).reduce(1,(x,y)->x\*y));  
 *//But for Factorial of 50 it will exceed Long max value so we can use BigInteger* System.*out*.println(LongStream.*rangeClosed*(1,50).mapToObj(BigInteger::*valueOf*).reduce(BigInteger.*ONE*,BigInteger::multiply));

Parallel Streams

It is meant for utilizing multiple cores of the processor. Normally any java code has one stream of processing, where it is executed sequentially. Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes

The order of execution, however, is not under our control.

Therefore, it is advisable to use parallel streams in cases where no matter what is the order of execution, the result is unaffected and the state of one element does not affect the other as well as the source of the data also remains unaffected.

Ex:

long t=System.*currentTimeMillis*();  
*//System.out.println(LongStream.rangeClosed(1,1000000000).sum()); //395 millisecs*System.*out*.println(LongStream.*rangeClosed*(1,1000000000).parallel().sum()); *//128 millisecs*System.*out*.println(System.*currentTimeMillis*()-t);

So by parallel streaming performance is increased

In below structured code there will be state, I,e sum is initialized and sum is appended by next int it keeps changing. So state is changing we cant run in 2 different cores

addSum(List<Integer> num)

{

int sum=0;

for(Integer n:num){

sum=sum+n;}

}

So in parallel stream, the stream is split into multiple cores and execute those and combine all the result. Ex For sum of 1000000 It will be divided 1 to 50000 in one core and 500000 to 100000 in other core parallely and then combine result.

We can achieve this by **adding the *parallel* method to a sequential stream or by creating a stream using the *parallelStream*method of a collection**:

List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);

listOfNumbers.parallelStream().forEach(number ->

System.out.println(number + " " + Thread.currentThread().getName())

);

Or

listOfNumbers.stream().parallel().forEach(number ->  
 System.*out*.println(number + " " + Thread.*currentThread*().getName())  
);

4 ForkJoinPool.commonPool-worker-3

2 ForkJoinPool.commonPool-worker-5

1 ForkJoinPool.commonPool-worker-7

3 main

FlatMap

flatMap() is the combination of a map and a flat operation i.e, it first applies map function and than flattens the result. [map()](https://www.geeksforgeeks.org/stream-map-java-examples/) is used for transformation only, but flatMap() is used for both transformation and flattening.

It means that in each iteration of each element the map() method creates a separate new stream. By using the flattening mechanism, it merges all streams into a single resultant stream. In short, it is used to convert a Stream of Stream into a list of values.

Ex1:

List<String> list1 = Arrays.*asList*("Geeks", "GFG", "GeeksforGeeks", "gfg");

*//split chars from all elements in list  
//when we split here it returns stream of string arrays  
//[["G","e","e","k","s"]["G","F","g"].....]  
//but we need all together so flat map converts stream of string arrays to single stream  
//Stream ["G","e","e","k","s","G","F","g",.......]*

list1.stream().map(course->course.split("")).flatMap(Arrays::*stream*).forEach(x->System.*out*.print(x+" "));

Output: G e e k s G F G G e e k s f o r G e e k s g f g

Ex2:

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

    {

        // Creating a list of Prime Numbers

        List<Integer> PrimeNumbers = Arrays.asList(5, 7, 11,13);

        // Creating a list of Odd Numbers

        List<Integer> OddNumbers = Arrays.asList(1, 3, 5);

        // Creating a list of Even Numbers

        List<Integer> EvenNumbers = Arrays.asList(2, 4, 6, 8);

        List<List<Integer>> listOfListofInts =

                Arrays.asList(PrimeNumbers, OddNumbers, EvenNumbers);

        System.out.println("The Structure before flattening is : " +

                                                  listOfListofInts);

        // Using flatMap for transformating and flattening.

        List<Integer> listofInts  = listOfListofInts.stream()

                                    .flatMap(list -> list.stream())

                                    .collect(Collectors.toList());

        System.out.println("The Structure after flattening is : " +

                                                         listofInts);

    }

Output

The Structure before flattening is : [[5, 7, 11, 13], [1, 3, 5], [2, 4, 6, 8]]

The Structure after flattening is : [5, 7, 11, 13, 1, 3, 5, 2, 4, 6, 8]

Features of Java8, DateTime API, Optional class, Abstract method, functional programming

### Can a functional interface extend/inherit another interface?

A functional interface cannot extend another interface with abstract methods as it will void the rule of one abstract method per functional interface.

It can extend other interfaces which do not have any abstract method and only have the default, static, another class is overridden, and normal methods

**Default & Static methods introduced in Java 8**

**Default Method:**

A method in the interface that has a predefined body is known as the default method. It uses the keyword default. default methods were introduced in Java 8 to have 'Backward Compatibility in case JDK modifies any interfaces. In case a new abstract method is added to the interface, all classes implementing the interface will break and will have to implement the new method. With default methods, there will not be any impact on the interface implementing classes. default methods can be overridden if needed in the implementation. Also, it does not qualify as synchronized or final.

**Static Method:**

Static methods, which contains method implementation is owned by the interface and is invoked using the name of the interface, it is suitable for defining the utility methods and cannot be overridden.

| **Sr. No.** | **Key** | **Static Interface Method** | **Default Method** |
| --- | --- | --- | --- |
| 1 | Basic | It is a static method which belongs to the interface only. We can write implementation of this method in interface itself | It is a method with default keyword and class can override this method |
| 2 | Method Invocation | Static method can invoke only on  interface class not on class. | It can be invoked on interface as well as class |
| 3 | Method Name | Interface and implementing class , both can have static method with the same name without overriding each other. | We can override the default method in implementing class |
| 4. | Use Case | It can be used as a utility method | It can be used to provide common functionality in all implementing classes |

Default Method (Observe these below mentioned points in below example or in Intellij exisitng code)

1. Now when a class will implement Interface1, it is not mandatory to provide implementation for default methods of interface. This feature will help us in extending interfaces with additional methods, all we need is to provide a default implementation.
2. We know that Java doesn’t allow us to extend multiple classes because it will result in the “Diamond Problem” where compiler can’t decide which superclass method to use. With the default methods, the diamond problem would arise for interfaces too. Because if a class is implementing both Interface1 and Interface2 and doesn’t implement the common default method, compiler can’t decide which one to chose. Extending multiple interfaces are an integral part of Java, you will find it in the core java classes as well as in most of the enterprise application and frameworks. So to make sure, this problem won’t occur in interfaces, it’s made mandatory to provide implementation for common default methods of interfaces. So if a class is implementing both the above interfaces, it will have to provide implementation for show1() method otherwise compiler will throw compile time error. **So if 2 same methods exists in 2 different interfaces and if we implements 2 interfaces in a class it is mandatory to implement those method in the clas or else error will occur.**

Static Methods:

1. Java interface static method is part of interface, we can’t use it for implementation class objects.
2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Java interface static method helps us in providing security by not allowing implementation classes to override them.

Ex:

**interface DefaultStaticExampleInterface1** {

default void show() {  
 System.*out*.println("In Java 8- default method - DefaultStaticExampleInterface");  
 }  
 default void show1() {  
 System.*out*.println("In Java 8- default method - DefaultStaticExampleInterface");  
 }  
 *// void show2(); //abstract method which don't have any body  
 //abstract methods should be mandatory to override in class but its not required for defualt methods* static void display() {  
 System.*out*.println("In DefaultStaticExampleInterface I");  
 }}

**interface DefaultStaticExampleInterface2**{  
 default void show1() {  
 System.*out*.println("In Java 8- default method - DefaultStaticExampleInterface");  
 }}

**public class DefaultStaticExampleClass implements DefaultStaticExampleInterface1,DefaultStaticExampleInterface2**{  
   
 static void display() {  
 System.*out*.println("In DefaultStaticExampleInterface I");  
 }  
 public static void main(String[] args) {  
 *// Call interface static method on Interface* DefaultStaticExampleInterface1.*display*();  
 DefaultStaticExampleClass defaultStaticExampleClass = new DefaultStaticExampleClass();  
  
 *// Call default method on Class* defaultStaticExampleClass.show();  
  
 DefaultStaticExampleClass.*display*();

*//To resolve diamond problem* defaultStaticExampleClass.show1();  
  
 @Override  
 public void show1() {  
 DefaultStaticExampleInterface1.super.show1();  
 //DefaultStaticExampleInterface2.super.show1();

}}

**Optional Class**

Optional is a container object which may or may not contain a non-null value. You must import ***java.util package*** to use this class. If a value is present, **isPresent()** will return true and **get()** will return the value.  Additional methods that depend on the presence or absence of a contained value are provided, such as **orElse()** which returns a default value if the value is not present. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run.

Ex:

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

    {

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

        String word = words[5].toLowerCase();

        System.out.print(word);

    }

**Output:**

Exception in thread "main" java.lang.NullPointerException

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

    {

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

        Optional<String> checkNull

            = Optional.ofNullable(words[5]);

**if** (checkNull.isPresent()) {

            String word = words[5].toLowerCase();

            System.out.print(word);

        }

**else**

            System.out.println("word is null");

    }

isPresent() method returns true if the wrapped value is not null.

### What are the advantages of using the Optional class?

Below are the main advantage of using the Optional class:

It encapsulates optional values, i.e., null or not-null values, which helps in avoiding null checks, which results in better, readable, and robust code It acts as a wrapper around the object and returns an object instead of a value, which can be used to avoid run-time NullPointerExceptions

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.

* **LocalDate/LocatTime**and **LocalDateTime API :**Use it when time zones are NOT required.
* **Zoned date-time API**: Use it when time zones are to be considered

Ex :

public static void main(String[] args) {  
 LocalDate localDate=LocalDate.*now*(); *//yyyy-MM-dd* LocalDateTime localDateTime=LocalDateTime.*now*();*//yyyy-MM-dd-HH-mm-ss.zzz (2020-02-20T12:06:03.015)* System.*out*.println(LocalDate.*of*(1998,12,01)); *//to customize  
  
 // to print in a particular format* DateTimeFormatter formatter=DateTimeFormatter.*ofPattern*("dd-MM HH:mm");  
 System.*out*.println( "in formatted manner "+localDateTime.format(formatter).toString());  
  
 *// printing months days and minutes* System.*out*.println("Month : "+localDateTime.getMonth()+" day : "+  
 localDateTime.getDayOfMonth()+" minutes : "+localDateTime.getMinute());  
 System.*out*.println(localDateTime);  
  
 *//To print Next Friday* System.*out*.println(localDate.with(TemporalAdjusters.*next*(DayOfWeek.*FRIDAY*)));  
 *//last day of this month* System.*out*.println(localDate.with(TemporalAdjusters.*lastDayOfMonth*()));

*//To print second saturday in this year*

LocalDate firstInYear = LocalDate.*of*(  
 localDate.getYear(), localDate.getMonth() , 1);  
 LocalDate secondSaturday  
 = firstInYear  
 .with(TemporalAdjusters.*nextOrSame*(  
 DayOfWeek.*SATURDAY*))  
 .with(TemporalAdjusters.*next*(  
 DayOfWeek.*SATURDAY*));  
  
 *// print date of second Saturday of this year* System.*out*.println("Second saturday is on : "  
 + secondSaturday);  
  
 ZonedDateTime zonedDateTime=ZonedDateTime.*now*();  
 System.*out*.println(zonedDateTime);  
  
 ZoneId zoneId=ZoneId.*of*("Asia/Tokyo");  
 *// getting time zone of specific place we use withZoneSameInstant(): it is  
 // used to return a copy of this date-time with a different time-zone,  
 // retaining the instant.* ZonedDateTime tokyoZone=zonedDateTime.withZoneSameInstant(zoneId);  
  
 System.*out*.println(tokyoZone);  
  
 DateTimeFormatter format =  
 DateTimeFormatter.*ofPattern*("dd-MM-yyyy HH:mm:ss");  
  
 String formatedDateTime = tokyoZone.format(format);  
  
 System.*out*.println("formatted tokyo time zone "+  
 formatedDateTime);  
}