# For each Method:

Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get [ConcurrentModificationException](https://www.journaldev.com/378/java-util-concurrentmodificationexception) if iterator is not used properly.

Java 8 has introduced *forEach* method in java.lang.Iterable interface so that while writing code we focus on business logic only. *forEach* method takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse. Let’s see forEach usage with simple example.

package com.journaldev.java8.foreach;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.function.Consumer;

import java.lang.Integer;

public class Java8ForEachExample {

public static void main(String[] args) {

//creating sample Collection

List<Integer> myList = new ArrayList<Integer>();

for(int i=0; i<10; i++) myList.add(i);

//traversing using Iterator

Iterator<Integer> it = myList.iterator();

while(it.hasNext()){

Integer i = it.next();

System.out.println("Iterator Value::"+i);

}

//traversing through forEach method of Iterable with anonymous class

myList.forEach(new Consumer<Integer>() {

public void accept(Integer t) {

System.out.println("forEach anonymous class Value::"+t);

}

});

//traversing with Consumer interface implementation

MyConsumer action = new MyConsumer();

myList.forEach(action);

}

}

//Consumer implementation that can be reused

class MyConsumer implements Consumer<Integer>{

public void accept(Integer t) {

System.out.println("Consumer impl Value::"+t);

}

}

The number of lines might increase but forEach method helps in having the logic for iteration and business logic at separate place resulting in higher separation of concern and cleaner code.

# Default method and static method in interface

Designing interfaces have always been a tough job because if we want to add additional methods in the interfaces, it will require change in all the implementing classes. As interface grows old, the number of classes implementing it might grow to an extent that it’s not possible to extend interfaces. That’s why when designing an application, most of the frameworks provide a base implementation class and then we extend it and override methods that are applicable for our application.

Default method:

For creating a default method in java interface, we need to use “**default**” keyword with the method signature. For example,

package com.journaldev.java8.defaultmethod;

public interface Interface1 {

void method1(String str);

default void log(String str){

System.out.println("I1 logging::"+str);

}

}

Notice that log(String str) is the default method in the Interface1. 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.

Let’s say we have another interface with following methods:

package com.journaldev.java8.defaultmethod;

public interface Interface2 {

void method2();

default void log(String str){

System.out.println("I2 logging::"+str);

}

}

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 log() method otherwise compiler will throw compile time error.

A simple class that is implementing both Interface1 and Interface2 will be:

package com.journaldev.java8.defaultmethod;

public class MyClass implements Interface1, Interface2 {

@Override

public void method2() {

}

@Override

public void method1(String str) {

}

@Override

public void log(String str){

System.out.println("MyClass logging::"+str);

Interface1.print("abc");

}

}

Important points about java interface default methods:

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.
5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
7. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

## Java Interface Static Method

Java interface static method is similar to default method except that we can’t override them in the implementation classes. This feature helps us in avoiding undesired results incase of poor implementation in implementation classes. Let’s look into this with a simple example.

package com.journaldev.java8.staticmethod;

public interface MyData {

default void print(String str) {

if (!isNull(str))

System.out.println("MyData Print::" + str);

}

static boolean isNull(String str) {

System.out.println("Interface Null Check");

return str == null ? true : "".equals(str) ? true : false;

}

}

Now let’s see an implementation class that is having isNull() method with poor implementation.

package com.journaldev.java8.staticmethod;

public class MyDataImpl implements MyData {

public boolean isNull(String str) {

System.out.println("Impl Null Check");

return str == null ? true : false;

}

public static void main(String args[]){

MyDataImpl obj = new MyDataImpl();

obj.print("");

obj.isNull("abc");

}

}

Note that isNull(String str) is a simple class method, it’s not overriding the interface method. For example, if we will add [@Override annotation](https://www.journaldev.com/817/java-override-method-overriding) to the isNull() method, it will result in compiler error.

Now when we will run the application, we get following output.

Interface Null Check

Impl Null Check

If we make the interface method from static to default, we will get following output.

Impl Null Check

MyData Print::

Impl Null Check

Java interface static method is visible to interface methods only, if we remove the isNull() method from the MyDataImpl class, we won’t be able to use it for the MyDataImpl object. However like other static methods, we can use interface static methods using class name. For example, a valid statement will be:

boolean result = MyData.isNull("abc");

Important points about java interface static method:

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.
4. We can’t define interface static method for Object class methods, we will get compiler error as “This static method cannot hide the instance method from Object”. This is because it’s not allowed in java, since Object is the base class for all the classes and we can’t have one class level static method and another instance method with same signature.
5. We can use java interface static methods to remove utility classes such as Collections and move all of it’s static methods to the corresponding interface, that would be easy to find and use.

### Java Functional Interfaces

Before I conclude the post, I would like to provide a brief introduction to Functional interfaces. An interface with exactly one abstract method is known as Functional Interface.

A new annotation @FunctionalInterface has been introduced to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It’s optional but good practice to use it.

Functional interfaces are long awaited and much sought out feature of Java 8 because it enables us to use **lambda expressions** to instantiate them. A new package java.util.function with bunch of functional interfaces are added to provide target types for lambda expressions and method references

# Lambda Expression

If we look into some other programming languages such as C++, JavaScript; they are called **functional programming language** because we can write functions and use them when required. Some of these languages support Object Oriented Programming as well as Functional Programming.

Being object oriented is not bad, but it brings a lot of verbosity to the program. For example, let’s say we have to create an instance of Runnable. Usually we do it using anonymous classes like below.

Runnable r = new Runnable(){

@Override

public void run() {

System.out.println("My Runnable");

}};

If you look at the above code, the actual part that is of use is the code inside run() method. Rest all of the code is because of the way java programs are structured.

Java 8 Functional Interfaces and Lambda Expressions help us in writing smaller and cleaner code by removing a lot of boiler-plate code.

## Functional Interface:

An interface with exactly one abstract method is called Functional Interface. @FunctionalInterface annotation is added so that we can mark an interface as functional interface.

It is not mandatory to use it, but it’s best practice to use it with functional interfaces to avoid addition of extra methods accidentally. If the interface is annotated with @FunctionalInterface annotation and we try to have more than one abstract method, it throws compiler error.

The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.

Java 8 Collections API has been rewritten and new Stream API is introduced that uses a lot of functional interfaces. Java 8 has defined a lot of functional interfaces in java.util.function package. Some of the useful java 8 functional interfaces are Consumer, Supplier, Function and Predicate.

You can find more detail about them in [Java 8 Stream Example](https://www.journaldev.com/2774/java-8-stream).

java.lang.Runnable is a great example of functional interface with single abstract method run().

Below code snippet provides some guidance for functional interfaces:

interface Foo { boolean equals(Object obj); }

// Not functional because equals is already an implicit member (Object class)

interface Comparator<T> {

boolean equals(Object obj);

int compare(T o1, T o2);

}

// Functional because Comparator has only one abstract non-Object method

interface Foo {

int m();

Object clone();

}

// Not functional because method Object.clone is not public

interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<String> arg); }

interface Z extends X, Y {}

// Functional: two methods, but they have the same signature

interface X { Iterable m(Iterable<String> arg); }

interface Y { Iterable<String> m(Iterable arg); }

interface Z extends X, Y {}

// Functional: Y.m is a subsignature & return-type-substitutable

interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<Integer> arg); }

interface Z extends X, Y {}

// Not functional: No method has a subsignature of all abstract methods

interface X { int m(Iterable<String> arg, Class c); }

interface Y { int m(Iterable arg, Class<?> c); }

interface Z extends X, Y {}

// Not functional: No method has a subsignature of all abstract methods

interface X { long m(); }

interface Y { int m(); }

interface Z extends X, Y {}

// Compiler error: no method is return type substitutable

interface Foo<T> { void m(T arg); }

interface Bar<T> { void m(T arg); }

interface FooBar<X, Y> extends Foo<X>, Bar<Y> {}

// Compiler error: different signatures, same erasure

### Lambda Expression

Lambda Expression are the way through which we can visualize **functional programming** in the java object oriented world. Objects are the base of java programming language and we can never have a function without an Object, that’s why Java language provide support for using lambda expressions only with functional interfaces.

Since there is only one abstract function in the functional interfaces, there is no confusion in applying the lambda expression to the method. Lambda Expressions syntax is **(argument) -> (body)**. Now let’s see how we can write above anonymous Runnable using lambda expression.

Runnable r1 = () -> System.out.println("My Runnable");

Let’s try to understand what is happening in the lambda expression above.

* Runnable is a functional interface, that’s why we can use lambda expression to create it’s instance.
* Since run() method takes no argument, our lambda expression also have no argument.
* Just like if-else blocks, we can avoid curly braces ({}) since we have a single statement in the method body. For multiple statements, we would have to use curly braces like any other methods.

### Why do we need Lambda Expression

1. **Reduced Lines of Code**  
   One of the clear benefit of using lambda expression is that the amount of code is reduced, we have already seen that how easily we can create instance of a functional interface using lambda expression rather than using anonymous class.
2. **Sequential and Parallel Execution Support**

Another benefit of using lambda expression is that we can benefit from the Stream API sequential and parallel operations support.

To explain this, let’s take a simple example where we need to write a method to test if a number passed is prime number or not.

Traditionally we would write it’s code like below. The code is not fully optimized but good for example purpose, so bear with me on this.

//Traditional approach

private static boolean isPrime(int number) {

if(number < 2) return false;

for(int i=2; i<number; i++){

if(number % i == 0) return false;

}

return true;

}

The problem with above code is that it’s sequential in nature, if the number is very huge then it will take significant amount of time. Another problem with code is that there are so many exit points and it’s not readable. Let’s see how we can write the same method using lambda expressions and stream API.

//Declarative approach

private static boolean isPrime(int number) {

return number > 1

&& IntStream.range(2, number).noneMatch(

index -> number % index == 0);

}

IntStream is a sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream.

For more readability, we can also write the method like below.

private static boolean isPrime(int number) {

IntPredicate isDivisible = index -> number % index == 0;

return number > 1

&& IntStream.range(2, number).noneMatch(

isDivisible);

}

1. If you are not familiar with IntStream, it’s range() method returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

noneMatch() method returns whether no elements of this stream match the provided predicate. It may not evaluate the predicate on all elements if not necessary for determining the result.

1. **Passing Behaviors into methods**

Let’s see how we can use lambda expressions to pass behavior of a method with a simple example. Let’s say we have to write a method to sum the numbers in a list if they match a given criteria. We can use Predicate and write a method like below.

public static int sumWithCondition(List<Integer> numbers, Predicate<Integer> predicate) {

return numbers.parallelStream()

.filter(predicate)

.mapToInt(i -> i)

.sum();

}

1. Sample usage:
2. //sum of all numbers
3. sumWithCondition(numbers, n -> true)
4. //sum of all even numbers
5. sumWithCondition(numbers, i -> i%2==0)
6. //sum of all numbers greater than 5
7. sumWithCondition(numbers, i -> i>5)
8. **Higher Efficiency with Laziness**

One more advantage of using lambda expression is the lazy evaluation, for example let’s say we need to write a method to find out the maximum odd number in the range 3 to 11 and return square of it.

Usually we will write code for this method like this:

private static int findSquareOfMaxOdd(List<Integer> numbers) {

int max = 0;

for (int i : numbers) {

if (i % 2 != 0 && i > 3 && i < 11 && i > max) {

max = i;

}

}

return max \* max;

}Above program will always run in sequential order but we can use Stream API to achieve this and get benefit of Laziness-seeking. Let’s see how we can rewrite this code in functional programming way using Stream API and lambda expressions.

public static int findSquareOfMaxOdd(List<Integer> numbers) {

return numbers.stream()

.filter(NumberTest::isOdd) //Predicate is functional interface and

.filter(NumberTest::isGreaterThan3) // we are using lambdas to initialize it

.filter(NumberTest::isLessThan11) // rather than anonymous inner classes

.max(Comparator.naturalOrder())

.map(i -> i \* i)

.get();

}

public static boolean isOdd(int i) {

return i % 2 != 0;

}

public static boolean isGreaterThan3(int i){

return i > 3;

}

public static boolean isLessThan11(int i){

return i < 11;

}

If you are surprised with the double colon (::) operator, it’s introduced in Java 8 and used for **method references**. Java Compiler takes care of mapping the arguments to the called method. It’s short form of lambda expressions i -> isGreaterThan3(i) or i -> NumberTest.isGreaterThan3(i).

### Lambda Expression Examples

Below I am providing some code snippets for lambda expressions with small comments explaining them.

() -> {} // No parameters; void result

() -> 42 // No parameters, expression body

() -> null // No parameters, expression body

() -> { return 42; } // No parameters, block body with return

() -> { System.gc(); } // No parameters, void block body

// Complex block body with multiple returns

() -> {

if (true) return 10;

else {

int result = 15;

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

result \*= i;

return result;

}

}

(int x) -> x+1 // Single declared-type argument

(int x) -> { return x+1; } // same as above

(x) -> x+1 // Single inferred-type argument, same as below

x -> x+1 // Parenthesis optional for single inferred-type case

(String s) -> s.length() // Single declared-type argument

(Thread t) -> { t.start(); } // Single declared-type argument

s -> s.length() // Single inferred-type argument

t -> { t.start(); } // Single inferred-type argument

(int x, int y) -> x+y // Multiple declared-type parameters

(x,y) -> x+y // Multiple inferred-type parameters

(x, final y) -> x+y // Illegal: can't modify inferred-type parameters

(x, int y) -> x+y // Illegal: can't mix inferred and declared types

### Method and Constructor References

A method reference is used to refer to a method without invoking it; a constructor reference is similarly used to refer to a constructor without creating a new instance of the named class or array type.

Examples of method and constructor references:

System::getProperty

System.out::println

"abc"::length

ArrayList::new

int[]::new

That’s all for Java 8 Functional Interfaces and Lambda Expression Tutorial. I would strongly suggest to look into using it because this syntax is new to Java and it will take some time to grasp it.

# STREAM IN JAVA8

### Java Stream

Before we look into Java Stream API Examples, let’s see why it was required. Suppose we want to iterate over a list of integers and find out sum of all the integers greater than 10.

Prior to Java 8, the approach to do it would be:

private static int sumIterator(List<Integer> list) {

Iterator<Integer> it = list.iterator();

int sum = 0;

while (it.hasNext()) {

int num = it.next();

if (num > 10) {

sum += num;

}

}

return sum;

}

There are three major problems with the above approach:

1. We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called **external iteration** because client program is handling the algorithm to iterate over the list.
2. The program is sequential in nature, there is no way we can do this in parallel easily.
3. There is a lot of code to do even a simple task.

To overcome all the above shortcomings, Java 8 Stream API was introduced. We can use Java Stream API to implement **internal iteration**, that is better because java framework is in control of the iteration.

**Internal iteration** provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them. Let’s see how can we write above logic in a single line statement using Java Streams.

private static int sumStream(List<Integer> list) {

return list.stream().filter(i -> i > 10).mapToInt(i -> i).sum();

}

Notice that above program utilizes java framework iteration strategy, filtering and mapping methods and would increase efficiency.

First of all we will look into the core concepts of Java 8 Stream API and then we will go through some examples for understanding most commonly used methods.

### Collections and Java Stream

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is a data structure that is computed on-demand.

Java Stream doesn’t store data, it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations. Such as we can create a stream from the list and filter it based on a condition.

Java Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expression. As you can see in the above example that using lambda expressions make our code readable and short.

Java 8 Stream internal iteration principle helps in achieving lazy-seeking in some of the stream operations. For example filtering, mapping, or duplicate removal can be implemented lazily, allowing higher performance and scope for optimization.

Java Streams are consumable, so there is no way to create a reference to stream for future usage. Since the data is on-demand, it’s not possible to reuse the same stream multiple times.

Java 8 Stream support sequential as well as parallel processing, parallel processing can be very helpful in achieving high performance for large collections.

All the Java Stream API interfaces and classes are in the java.util.stream package. Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for primitive types – IntStream, LongStream and DoubleStream.

### Functional Interfaces in Java 8 Stream

Some of the commonly used functional interfaces in the Java 8 Stream API methods are:

1. **Function and BiFunction**: Function represents a function that takes one type of argument and returns another type of argument. Function<T, R> is the generic form where T is the type of the input to the function and R is the type of the result of the function.

For handling primitive types, there are specific Function interfaces – ToIntFunction, ToLongFunction, ToDoubleFunction, ToIntBiFunction, ToLongBiFunction, ToDoubleBiFunction, LongToIntFunction, LongToDoubleFunction, IntToLongFunction, IntToDoubleFunction etc.

Some of the Stream methods where Function or it’s primitive specialization is used are:

* + <R> Stream<R> map(Function<? super T, ? extends R> mapper)
  + IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.
  + IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double
  + <A> A[] toArray(IntFunction<A[]> generator)
  + <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

1. **Predicate and BiPredicate**: It represents a predicate against which elements of the stream are tested. This is used to filter elements from the java stream. Just like Function, there are primitive specific interfaces for int, long and double.

Some of the Stream methods where Predicate or BiPredicate specializations are used are:

* + Stream<T> filter(Predicate<? super T> predicate)
  + boolean anyMatch(Predicate<? super T> predicate)
  + boolean allMatch(Predicate<? super T> predicate)
  + boolean noneMatch(Predicate<? super T> predicate)

1. **Consumer and BiConsumer**: It represents an operation that accepts a single input argument and returns no result. It can be used to perform some action on all the elements of the java stream.

Some of the Java 8 Stream methods where Consumer, BiConsumer or it’s primitive specialization interfaces are used are:

* + Stream<T> peek(Consumer<? super T> action)
  + void forEach(Consumer<? super T> action)
  + void forEachOrdered(Consumer<? super T> action)

1. **Supplier**: Supplier represent an operation through which we can generate new values in the stream. Some of the methods in Stream that takes Supplier argument are:
   * public static<T> Stream<T> generate(Supplier<T> s)
   * <R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

### java.util.Optional

[Java Optional](https://www.journaldev.com/16709/java-optional) is a container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value. Stream terminal operations return Optional object. Some of these methods are:

* Optional<T> reduce(BinaryOperator<T> accumulator)
* Optional<T> min(Comparator<? super T> comparator)
* Optional<T> max(Comparator<? super T> comparator)
* Optional<T> findFirst()
* Optional<T> findAny()

### java.util.Spliterator

For supporting parallel execution in Java 8 Stream API, Spliterator interface is used. Spliterator trySplit method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

### Java Stream Intermediate and Terminal Operations

Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations. Commonly used intermediate operations are filter and map.

Java 8 Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can’t use stream. Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result. Commonly used terminal methods are forEach, toArray, min, max, findFirst, anyMatch, allMatch etc. You can identify terminal methods from the return type, they will never return a Stream.

### Java Stream Short Circuiting Operations

An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example limit() and skip() are two short circuiting intermediate operations.

A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example anyMatch, allMatch, noneMatch, findFirst and findAny are short circuiting terminal operations.

### Java Stream Examples

I have covered almost all the important parts of the Java 8 Stream API. It’s exciting to use this new API features and let’s see it in action with some java stream examples.

#### Creating Java Streams

There are several ways through which we can create a java stream from array and collections. Let’s look into these with simple examples.

1. We can use Stream.of() to create a stream from similar type of data. For example, we can create Java Stream of integers from a group of int or Integer objects.
2. Stream<Integer> stream = Stream.of(1,2,3,4);
3. We can use Stream.of() with an array of Objects to return the stream. Note that it doesn’t support autoboxing, so we can’t pass primitive type array.
4. Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4});
5. //works fine
6. Stream<Integer> stream1 = Stream.of(new int[]{1,2,3,4});
7. //Compile time error, Type mismatch: cannot convert from Stream<int[]> to Stream<Integer>
8. We can use Collection stream() to create sequential stream and parallelStream() to create parallel stream.
9. List<Integer> myList = new ArrayList<>();
10. for(int i=0; i<100; i++) myList.add(i);
12. //sequential stream
13. Stream<Integer> sequentialStream = myList.stream();
15. //parallel stream
16. Stream<Integer> parallelStream = myList.parallelStream();
17. We can use Stream.generate() and Stream.iterate() methods to create Stream.
18. Stream<String> stream1 = Stream.generate(() -> {return "abc";});
19. Stream<String> stream2 = Stream.iterate("abc", (i) -> i);
20. Using Arrays.stream() and String.chars() methods.
21. LongStream is = Arrays.stream(new long[]{1,2,3,4});
22. IntStream is2 = "abc".chars();

#### Converting Java Stream to Collection or Array

There are several ways through which we can get a Collection or Array from a java Stream.

1. We can use java Stream collect() method to get List, Map or Set from stream.
2. Stream<Integer> intStream = Stream.of(1,2,3,4);
3. List<Integer> intList = intStream.collect(Collectors.toList());
4. System.out.println(intList); //prints [1, 2, 3, 4]
5. intStream = Stream.of(1,2,3,4); //stream is closed, so we need to create it again
6. Map<Integer,Integer> intMap = intStream.collect(Collectors.toMap(i -> i, i -> i+10));
7. System.out.println(intMap); //prints {1=11, 2=12, 3=13, 4=14}
8. We can use stream toArray() method to create an array from the stream.
9. Stream<Integer> intStream = Stream.of(1,2,3,4);
10. Integer[] intArray = intStream.toArray(Integer[]::new);
11. System.out.println(Arrays.toString(intArray)); //prints [1, 2, 3, 4]

### Java Stream Intermediate Operations

Let’s look into commonly used java Stream intermediate operations example.

1. **Stream filter() example**: We can use filter() method to test stream elements for a condition and generate filtered list.
2. List<Integer> myList = new ArrayList<>();
3. for(int i=0; i<100; i++) myList.add(i);
4. Stream<Integer> sequentialStream = myList.stream();
5. Stream<Integer> highNums = sequentialStream.filter(p -> p > 90); //filter numbers greater than 90
6. System.out.print("High Nums greater than 90=");
7. highNums.forEach(p -> System.out.print(p+" "));
8. //prints "High Nums greater than 90=91 92 93 94 95 96 97 98 99 "
9. **Stream map() example**: We can use map() to apply functions to an stream. Let’s see how we can use it to apply upper case function to a list of Strings.
10. Stream<String> names = Stream.of("aBc", "d", "ef");
11. System.out.println(names.map(s -> {
12. return s.toUpperCase();
13. }).collect(Collectors.toList()));
14. //prints [ABC, D, EF]
15. **Stream sorted() example**: We can use sorted() to sort the stream elements by passing Comparator argument.
16. Stream<String> names2 = Stream.of("aBc", "d", "ef", "123456");
17. List<String> reverseSorted = names2.sorted(Comparator.reverseOrder()).collect(Collectors.toList());
18. System.out.println(reverseSorted); // [ef, d, aBc, 123456]
19. Stream<String> names3 = Stream.of("aBc", "d", "ef", "123456");
20. List<String> naturalSorted = names3.sorted().collect(Collectors.toList());
21. System.out.println(naturalSorted); //[123456, aBc, d, ef]
22. **Stream flatMap() example**: We can use flatMap() to create a stream from the stream of list. Let’s see a simple example to clear this doubt.
23. Stream<List<String>> namesOriginalList = Stream.of(
24. Arrays.asList("Pankaj"),
25. Arrays.asList("David", "Lisa"),
26. Arrays.asList("Amit"));
27. //flat the stream from List<String> to String stream
28. Stream<String> flatStream = namesOriginalList
29. .flatMap(strList -> strList.stream());
30. flatStream.forEach(System.out::println);

### Java Stream Terminal Operations

Let’s look at some of the java stream terminal operations example.

1. **Stream reduce() example**: We can use reduce() to perform a reduction on the elements of the stream, using an associative accumulation function, and return an Optional. Let’s see how we can use it multiply the integers in a stream.
2. Stream<Integer> numbers = Stream.of(1,2,3,4,5);
4. Optional<Integer> intOptional = numbers.reduce((i,j) -> {return i\*j;});
5. if(intOptional.isPresent()) System.out.println("Multiplication = "+intOptional.get()); //120
6. **Stream count() example**: We can use this terminal operation to count the number of items in the stream.
7. Stream<Integer> numbers1 = Stream.of(1,2,3,4,5);
9. System.out.println("Number of elements in stream="+numbers1.count()); //5
10. **Stream forEach() example**: This can be used for iterating over the stream. We can use this in place of iterator. Let’s see how to use it for printing all the elements of the stream.
11. Stream<Integer> numbers2 = Stream.of(1,2,3,4,5);
12. numbers2.forEach(i -> System.out.print(i+",")); //1,2,3,4,5,
13. **Stream match() examples**: Let’s see some of the examples for matching methods in Stream API.
14. Stream<Integer> numbers3 = Stream.of(1,2,3,4,5);
15. System.out.println("Stream contains 4? "+numbers3.anyMatch(i -> i==4));
16. //Stream contains 4? true
17. Stream<Integer> numbers4 = Stream.of(1,2,3,4,5);
18. System.out.println("Stream contains all elements less than 10? "+numbers4.allMatch(i -> i<10));
19. //Stream contains all elements less than 10? true
20. Stream<Integer> numbers5 = Stream.of(1,2,3,4,5);
21. System.out.println("Stream doesn't contain 10? "+numbers5.noneMatch(i -> i==10));
22. //Stream doesn't contain 10? true
23. **Stream findFirst() example**: This is a short circuiting terminal operation, let’s see how we can use it to find the first string from a stream starting with D.
24. Stream<String> names4 = Stream.of("Pankaj","Amit","David", "Lisa");
25. Optional<String> firstNameWithD = names4.filter(i -> i.startsWith("D")).findFirst();
26. if(firstNameWithD.isPresent()){
27. System.out.println("First Name starting with D="+firstNameWithD.get()); //David
28. }

### Java 8 Stream API Limitations

Java 8 Stream API brings a lot of new stuffs to work with list and arrays, but it has some limitations too.

1. **Stateless lambda expressions**: If you are using parallel stream and lambda expressions are stateful, it can result in random responses. Let’s see it with a simple program.

StatefulParallelStream.java

package com.journaldev.java8.stream;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class StatefulParallelStream {

public static void main(String[] args) {

List<Integer> ss = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);

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

Stream<Integer> stream = ss.parallelStream();

stream.map(s -> {

synchronized (result) {

if (result.size() < 10) {

result.add(s);

}

}

return s;

}).forEach( e -> {});

System.out.println(result);

}

}

If we run above program, you will get different results because it depends on the way stream is getting iterated and we don’t have any order defined for parallel processing. If we use sequential stream, then this problem will not arise.

1. Once a Stream is consumed, it can’t be used later on. As you can see in above examples that every time I am creating a stream.
2. There are a lot of methods in Stream API and the most confusing part is the overloaded methods. It makes the learning curve time taking.

# JAVA date time API

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

Before we start looking at the Java 8 Date Time API, let’s see why do we need a new API for this. There have been several problems with the existing date and time related classes in java, some of them are:

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](https://www.joda.org/joda-time/" \t "_blank) played a key role as a quality replacement for Java Date Time requirements.

## Java 8 Date

[](https://cdn.journaldev.com/wp-content/uploads/2014/04/java-8-date-localdate-localdatetime-instant.jpg)

Java 8 Date Time API is [JSR-310](https://jcp.org/en/jsr/detail?id=310) implementation. It is designed to overcome all the flaws in the legacy date time implementations. Some of the design principles of new Date Time API are:

1. **Immutability**: All the classes in the new Date Time API are immutable and good for multithreaded environments.
2. **Separation of Concerns**: The new API separates clearly between human readable date time and machine time (unix timestamp). It defines separate classes for Date, Time, DateTime, Timestamp, Timezone etc.
3. **Clarity**: The methods are clearly defined and perform the same action in all the classes. For example, to get the current instance we have now() method. There are format() and parse() methods defined in all these classes rather than having a separate class for them.

All the classes use [Factory Pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java) and [Strategy Pattern](https://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial) for better handling. Once you have used the methods in one of the class, working with other classes won’t be hard.

1. **Utility operations**: All the new Date Time API classes comes with methods to perform common tasks, such as plus, minus, format, parsing, getting separate part in date/time etc.
2. **Extendable**: The new Date Time API works on ISO-8601 calendar system but we can use it with other non ISO calendars as well.

### Java 8 Date Time API Packages

Java 8 Date Time API consists of following 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 Date Time API Examples

We have looked into most of the important parts of Java Date Time API. It’s time now to look into most important classes of Date Time API with examples.

### LocalDate

LocalDate is an [immutable class](https://www.journaldev.com/129/how-to-create-immutable-class-in-java) that represents Date with default format of yyyy-MM-dd. We can use now() method to get the current date. We can also provide input arguments for year, month and date to create LocalDate instance. This class provides overloaded method for now() where we can pass ZoneId for getting date in specific time zone. This class provides the same functionality as java.sql.Date. Let’s look at a simple example for it’s usage.

package com.journaldev.java8.time;

import java.time.LocalDate;

import java.time.Month;

import java.time.ZoneId;

/\*\*

\* LocalDate Examples

\* @author pankaj

\*

\*/

public class LocalDateExample {

public static void main(String[] args) {

//Current Date

LocalDate today = LocalDate.now();

System.out.println("Current Date="+today);

//Creating LocalDate by providing input arguments

LocalDate firstDay\_2014 = LocalDate.of(2014, Month.JANUARY, 1);

System.out.println("Specific Date="+firstDay\_2014);

//Try creating date by providing invalid inputs

//LocalDate feb29\_2014 = LocalDate.of(2014, Month.FEBRUARY, 29);

//Exception in thread "main" java.time.DateTimeException:

//Invalid date 'February 29' as '2014' is not a leap year

//Current date in "Asia/Kolkata", you can get it from ZoneId javadoc

LocalDate todayKolkata = LocalDate.now(ZoneId.of("Asia/Kolkata"));

System.out.println("Current Date in IST="+todayKolkata);

//java.time.zone.ZoneRulesException: Unknown time-zone ID: IST

//LocalDate todayIST = LocalDate.now(ZoneId.of("IST"));

//Getting date from the base date i.e 01/01/1970

LocalDate dateFromBase = LocalDate.ofEpochDay(365);

System.out.println("365th day from base date= "+dateFromBase);

LocalDate hundredDay2014 = LocalDate.ofYearDay(2014, 100);

System.out.println("100th day of 2014="+hundredDay2014);

}

}

LocalDate methods explanation is provided in comments, when we run this program, we get following output.

Current Date=2014-04-28

Specific Date=2014-01-01

Current Date in IST=2014-04-29

365th day from base date= 1971-01-01

100th day of 2014=2014-04-10

### LocalTime

LocalTime is an immutable class whose instance represents a time in the human readable format. It’s default format is hh:mm:ss.zzz. Just like LocalDate, this class provides time zone support and creating instance by passing hour, minute and second as input arguments. Let’s look at it’s usage with a simple program.

package com.journaldev.java8.time;

import java.time.LocalTime;

import java.time.ZoneId;

/\*\*

\* LocalTime Examples

\* @author pankaj

\*

\*/

public class LocalTimeExample {

public static void main(String[] args) {

//Current Time

LocalTime time = LocalTime.now();

System.out.println("Current Time="+time);

//Creating LocalTime by providing input arguments

LocalTime specificTime = LocalTime.of(12,20,25,40);

System.out.println("Specific Time of Day="+specificTime);

//Try creating time by providing invalid inputs

//LocalTime invalidTime = LocalTime.of(25,20);

//Exception in thread "main" java.time.DateTimeException:

//Invalid value for HourOfDay (valid values 0 - 23): 25

//Current date in "Asia/Kolkata", you can get it from ZoneId javadoc

LocalTime timeKolkata = LocalTime.now(ZoneId.of("Asia/Kolkata"));

System.out.println("Current Time in IST="+timeKolkata);

//java.time.zone.ZoneRulesException: Unknown time-zone ID: IST

//LocalTime todayIST = LocalTime.now(ZoneId.of("IST"));

//Getting date from the base date i.e 01/01/1970

LocalTime specificSecondTime = LocalTime.ofSecondOfDay(10000);

System.out.println("10000th second time= "+specificSecondTime);

}

}

When we run above program for LocalTime examples, we get following output.

Current Time=15:51:45.240

Specific Time of Day=12:20:25.000000040

Current Time in IST=04:21:45.276

10000th second time= 02:46:40

### LocalDateTime

LocalDateTime is an immutable date-time object that represents a date-time, with default format as yyyy-MM-dd-HH-mm-ss.zzz. It provides a factory method that takes LocalDate and LocalTime input arguments to create LocalDateTime instance. Let’s look it’s usage with a simple example.

package com.journaldev.java8.time;

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.time.LocalTime;

import java.time.Month;

import java.time.ZoneId;

import java.time.ZoneOffset;

public class LocalDateTimeExample {

public static void main(String[] args) {

//Current Date

LocalDateTime today = LocalDateTime.now();

System.out.println("Current DateTime="+today);

//Current Date using LocalDate and LocalTime

today = LocalDateTime.of(LocalDate.now(), LocalTime.now());

System.out.println("Current DateTime="+today);

//Creating LocalDateTime by providing input arguments

LocalDateTime specificDate = LocalDateTime.of(2014, Month.JANUARY, 1, 10, 10, 30);

System.out.println("Specific Date="+specificDate);

//Try creating date by providing invalid inputs

//LocalDateTime feb29\_2014 = LocalDateTime.of(2014, Month.FEBRUARY, 28, 25,1,1);

//Exception in thread "main" java.time.DateTimeException:

//Invalid value for HourOfDay (valid values 0 - 23): 25

//Current date in "Asia/Kolkata", you can get it from ZoneId javadoc

LocalDateTime todayKolkata = LocalDateTime.now(ZoneId.of("Asia/Kolkata"));

System.out.println("Current Date in IST="+todayKolkata);

//java.time.zone.ZoneRulesException: Unknown time-zone ID: IST

//LocalDateTime todayIST = LocalDateTime.now(ZoneId.of("IST"));

//Getting date from the base date i.e 01/01/1970

LocalDateTime dateFromBase = LocalDateTime.ofEpochSecond(10000, 0, ZoneOffset.UTC);

System.out.println("10000th second time from 01/01/1970= "+dateFromBase);

}

}

In all the three examples, we have seen that if we provide invalid arguments for creating Date/Time, then it throws java.time.DateTimeException that is a RuntimeException, so we don’t need to explicitly catch it.

We have also seen that we can get Date/Time data by passing ZoneId, you can get the list of supported ZoneId values from it’s javadoc. When we run above class, we get following output.

Current DateTime=2014-04-28T16:00:49.455

Current DateTime=2014-04-28T16:00:49.493

Specific Date=2014-01-01T10:10:30

Current Date in IST=2014-04-29T04:30:49.493

10000th second time from 01/01/1970= 1970-01-01T02:46:40

### Instant

Instant class is used to work with machine readable time format, it stores date time in unix timestamp. Let’s see it’s usage with a simple program.

package com.journaldev.java8.time;

import java.time.Duration;

import java.time.Instant;

public class InstantExample {

public static void main(String[] args) {

//Current timestamp

Instant timestamp = Instant.now();

System.out.println("Current Timestamp = "+timestamp);

//Instant from timestamp

Instant specificTime = Instant.ofEpochMilli(timestamp.toEpochMilli());

System.out.println("Specific Time = "+specificTime);

//Duration example

Duration thirtyDay = Duration.ofDays(30);

System.out.println(thirtyDay);

}

}

Output of above program is:

Current Timestamp = 2014-04-28T23:20:08.489Z

Specific Time = 2014-04-28T23:20:08.489Z

PT720H

### Java 8 Date API Utilities

As mentioned earlier, most of the Date Time principle classes provide various utility methods such as plus/minus days, weeks, months etc. There are some other utility methods for adjusting the date using TemporalAdjuster and to calculate the period between two dates.

package com.journaldev.java8.time;

import java.time.LocalDate;

import java.time.LocalTime;

import java.time.Period;

import java.time.temporal.TemporalAdjusters;

public class DateAPIUtilities {

public static void main(String[] args) {

LocalDate today = LocalDate.now();

//Get the Year, check if it's leap year

System.out.println("Year "+today.getYear()+" is Leap Year? "+today.isLeapYear());

//Compare two LocalDate for before and after

System.out.println("Today is before 01/01/2015? "+today.isBefore(LocalDate.of(2015,1,1)));

//Create LocalDateTime from LocalDate

System.out.println("Current Time="+today.atTime(LocalTime.now()));

//plus and minus operations

System.out.println("10 days after today will be "+today.plusDays(10));

System.out.println("3 weeks after today will be "+today.plusWeeks(3));

System.out.println("20 months after today will be "+today.plusMonths(20));

System.out.println("10 days before today will be "+today.minusDays(10));

System.out.println("3 weeks before today will be "+today.minusWeeks(3));

System.out.println("20 months before today will be "+today.minusMonths(20));

//Temporal adjusters for adjusting the dates

System.out.println("First date of this month= "+today.with(TemporalAdjusters.firstDayOfMonth()));

LocalDate lastDayOfYear = today.with(TemporalAdjusters.lastDayOfYear());

System.out.println("Last date of this year= "+lastDayOfYear);

Period period = today.until(lastDayOfYear);

System.out.println("Period Format= "+period);

System.out.println("Months remaining in the year= "+period.getMonths());

}

}

Output of above program is:

Year 2014 is Leap Year? false

Today is before 01/01/2015? true

Current Time=2014-04-28T16:23:53.154

10 days after today will be 2014-05-08

3 weeks after today will be 2014-05-19

20 months after today will be 2015-12-28

10 days before today will be 2014-04-18

3 weeks before today will be 2014-04-07

20 months before today will be 2012-08-28

First date of this month= 2014-04-01

Last date of this year= 2014-12-31

Period Format= P8M3D

Months remaining in the year= 8

### Java 8 Date Parsing and Formatting

It’s very common to format date into different formats and then parse a String to get the Date Time objects. Let’s see it with simple examples.

package com.journaldev.java8.time;

import java.time.Instant;

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

public class DateParseFormatExample {

public static void main(String[] args) {

//Format examples

LocalDate date = LocalDate.now();

//default format

System.out.println("Default format of LocalDate="+date);

//specific format

System.out.println(date.format(DateTimeFormatter.ofPattern("d::MMM::uuuu")));

System.out.println(date.format(DateTimeFormatter.BASIC\_ISO\_DATE));

LocalDateTime dateTime = LocalDateTime.now();

//default format

System.out.println("Default format of LocalDateTime="+dateTime);

//specific format

System.out.println(dateTime.format(DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss")));

System.out.println(dateTime.format(DateTimeFormatter.BASIC\_ISO\_DATE));

Instant timestamp = Instant.now();

//default format

System.out.println("Default format of Instant="+timestamp);

//Parse examples

LocalDateTime dt = LocalDateTime.parse("27::Apr::2014 21::39::48",

DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss"));

System.out.println("Default format after parsing = "+dt);

}

}

When we run above program, we get following output.

Default format of LocalDate=2014-04-28

28::Apr::2014

20140428

Default format of LocalDateTime=2014-04-28T16:25:49.341

28::Apr::2014 16::25::49

20140428

Default format of Instant=2014-04-28T23:25:49.342Z

Default format after parsing = 2014-04-27T21:39:48

### Java 8 Date API Legacy Date Time Support

Legacy Date/Time classes are used in almost all the applications, so having backward compatibility is a must. That’s why there are several utility methods through which we can convert Legacy classes to new classes and vice versa. Let’s see this with a simple example.

package com.journaldev.java8.time;

import java.time.Instant;

import java.time.LocalDateTime;

import java.time.ZoneId;

import java.time.ZonedDateTime;

import java.util.Calendar;

import java.util.Date;

import java.util.GregorianCalendar;

import java.util.TimeZone;

public class DateAPILegacySupport {

public static void main(String[] args) {

//Date to Instant

Instant timestamp = new Date().toInstant();

//Now we can convert Instant to LocalDateTime or other similar classes

LocalDateTime date = LocalDateTime.ofInstant(timestamp,

ZoneId.of(ZoneId.SHORT\_IDS.get("PST")));

System.out.println("Date = "+date);

//Calendar to Instant

Instant time = Calendar.getInstance().toInstant();

System.out.println(time);

//TimeZone to ZoneId

ZoneId defaultZone = TimeZone.getDefault().toZoneId();

System.out.println(defaultZone);

//ZonedDateTime from specific Calendar

ZonedDateTime gregorianCalendarDateTime = new GregorianCalendar().toZonedDateTime();

System.out.println(gregorianCalendarDateTime);

//Date API to Legacy classes

Date dt = Date.from(Instant.now());

System.out.println(dt);

TimeZone tz = TimeZone.getTimeZone(defaultZone);

System.out.println(tz);

GregorianCalendar gc = GregorianCalendar.from(gregorianCalendarDateTime);

System.out.println(gc);

}

}

When we run above application, we get following output.

Date = 2014-04-28T16:28:54.340

2014-04-28T23:28:54.395Z

America/Los\_Angeles

2014-04-28T16:28:54.404-07:00[America/Los\_Angeles]

Mon Apr 28 16:28:54 PDT 2014

sun.util.calendar.ZoneInfo[id="America/Los\_Angeles",offset=-28800000,dstSavings=3600000,useDaylight=true,transitions=185,lastRule=java.util.SimpleTimeZone[id=America/Los\_Angeles,offset=-28800000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=3,startMonth=2,startDay=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]]

java.util.GregorianCalendar[time=1398727734404,areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="America/Los\_Angeles",offset=-28800000,dstSavings=3600000,useDaylight=true,transitions=185,lastRule=java.util.SimpleTimeZone[id=America/Los\_Angeles,offset=-28800000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=3,startMonth=2,startDay=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,YEAR=2014,MONTH=3,WEEK\_OF\_YEAR=18,WEEK\_OF\_MONTH=5,DAY\_OF\_MONTH=28,DAY\_OF\_YEAR=118,DAY\_OF\_WEEK=2,DAY\_OF\_WEEK\_IN\_MONTH=4,AM\_PM=1,HOUR=4,HOUR\_OF\_DAY=16,MINUTE=28,SECOND=54,MILLISECOND=404,ZONE\_OFFSET=-28800000,DST\_OFFSET=3600000]

As you can see that legacy TimeZone and GregorianCalendar classes toString() methods are too verbose and not user friendly.

That’s all for Java 8 Date Time API, I like this new API a lot. Some of the most used classes will be LocalDate and LocalDateTime for this new API. It’s very easy to work with and having similar methods that does a particular job makes it easy to find. It will take some time from moving legacy classes to new Date Time classes, but I believe it will be worthy of the time.

# Collection improvement:

### Collection API improvements

We have already seen forEach() method and Stream API for collections. Some new methods added in Collection API are:

* + Iterator default method forEachRemaining(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
  + Collection default method removeIf(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.
  + Collection spliterator() method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
  + Map replaceAll(), compute(), merge() methods.
  + Performance Improvement for HashMap class with Key Collisions

### Concurrency API improvements

Some important concurrent API enhancements are:

* + ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
  + CompletableFuture that may be explicitly completed (setting its value and status).
  + Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

### Java IO improvements

Some IO improvements known to me are:

* + Files.list(Path dir) that returns a lazily populated Stream, the elements of which are the entries in the directory.
  + Files.lines(Path path) that reads all lines from a file as a Stream.
  + Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
  + BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.

### Miscellaneous Core API improvements

Some misc API improvements that might come handy are:

* + [ThreadLocal](https://www.journaldev.com/1076/java-threadlocal-example) static method withInitial(Supplier supplier) to create instance easily.
  + [Comparator](https://www.journaldev.com/780/comparable-and-comparator-in-java-example) interface has been extended with a lot of default and static methods for natural ordering, reverse order etc.
  + min(), max() and sum() methods in Integer, Long and Double wrapper classes.
  + logicalAnd(), logicalOr() and logicalXor() methods in Boolean class.
  + [ZipFile](https://www.journaldev.com/957/java-zip-file-folder-example).stream() method to get an ordered Stream over the ZIP file entries. Entries appear in the Stream in the order they appear in the central directory of the ZIP file.
  + Several utility methods in Math class.
  + jjs command is added to invoke Nashorn Engine.
  + jdeps command is added to analyze class files
  + JDBC-ODBC Bridge has been removed.
  + PermGen memory space has been removed

That’s all for Java 8 features with example programs. If I have missed some important features of Java 8, please let me know through comments.

<https://www.oracle.com/technical-resources/articles/java/architect-streams-pt2.html>