Java 8

- Lambda Expressions
- Functional Interfaces
- Stream API
  - Intermediate
  - Terminal
  - Parallelism

  Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed:

- Date and Time API
- Interface Default Methods and Static Methods - default for backward compatibility.
- Spliterator
- Method and Constructor References
- Collections API Enhancements
- Concurrency Utils Enhancements
- Fork/Join Framework Enhancements
- Internal Iteration
- Parallel Array and Parallel Collection Operations
- Optional - special wrapper class u sed for expressing optionality
- Type Annotations and Repeatable Annotations
- Method Parameter Reflection
- Base64 Encoding and Decoding
- IO and NIO2 Enhancements
- Nashorn JavaScript Engine - java-based engine for executing and evaluating JavaScript code
- javac Enhancements
- JVM Changes
- Java 8 Compact Profiles: compact1,compact2,compact3
- JDBC 4.2
- JAXP 1.6
- Java DB 10.10

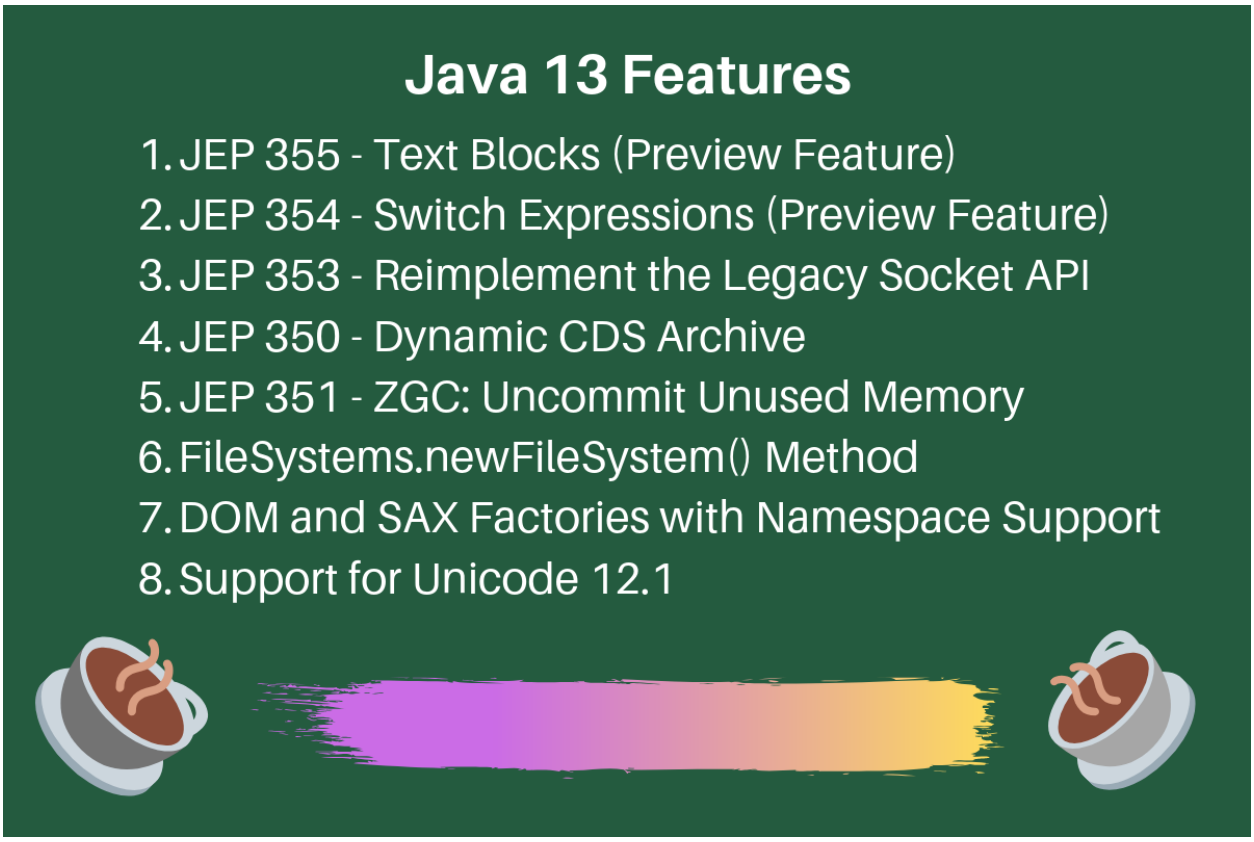
- Networking
- Security Change

Java 9

- Java 9 REPL (JShell) - It stands for Java Shell and also known as REPL (Read Evaluate Print Loop).
- Factory Methods for Immutable List, Set, Map and Map.Entry

```
List immutableList = List.of();
```

- Private methods in Interfaces
- Java 9 Module System
- Process API Improvements
- Try With Resources Improvement
- CompletableFuture API Improvements
- Reactive Streams
- Diamond Operator for Anonymous Inner Class
- Optional Class Improvements
- Stream API Improvements
- Enhanced @Deprecated annotation
- HTTP 2 Client - to support HTTP/2 protocol and WebSocket features.
- Multi-Resolution Image API
- Miscellaneous Java 9 Features



When do we go for Java 8 Stream API? Why do we need to use Java 8 Stream API in our projects?

When our Java project wants to perform the following operations, it's better to use Java 8 Stream API to get lot of benefits:

- When we want perform Database like Operations. For instance, we want perform groupby operation, orderby operation etc.
- When want to Perform operations Lazily.
- When we want to write Functional Style programming.
- When we want to perform Parallel Operations.
- When want to use Internal Iteration
- When we want to perform Pipelining operations.
- When we want to achieve better performance.

Explain Differences between Collection API and Stream API?

S.No.	Collection API	Stream API
1.	It's available since Java 1.2	It is introduced in Java SE8
2.	It is used to store Data(A set of Objects).	It is used to compute data(Computation on a set of Objects).
3.	We can use both Spliterator and Iterator to iterate elements. We can use forEach to performs an action for each element of this stream.	We can't use Spliterator or Iterator to iterate elements.
4.	It is used to store limited number of Elements.	It is used to store either Limited or Infinite Number of Elements.
5.	Typically, it uses External Iteration concept to iterate Elements such as Iterator.	Stream API uses internal iteration to iterate Elements, using forEach methods.
6.	Collection Object is constructed Eagerly.	Stream Object is constructed Lazily.
7.	We add elements to Collection object only after it is computed completely.	We can add elements to Stream Object without any prior computation. That means Stream objects are computed on-demand.
8.	We can iterate and consume elements from a Collection Object at any number of times.	We can iterate and consume elements from a Stream Object only once.

What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?

S.No.	Spliterator	Iterator
1.	It is introduced in Java SE 8.	It is available since Java 1.2.
2.	Splitable Iterator	Non-Splitable Iterator
3.	It is used in Stream API.	It is used for Collection API.
4.	It uses Internal Iteration concept to iterate Streams.	It uses External Iteration concept to iterate Collections.
5.	We can use Spliterator to iterate Streams in Parallel and Sequential order.	We can use Iterator to iterate Collections only in Sequential order.
6.	We can get Spliterator by calling spliterator() method on Stream Object.	We can get Iterator by calling iterator() method on Collection Object.
7.	Important Method: tryAdvance()	Important Methods: next(), hasNext()

What is Optional in Java 8? What is the use of Optional?Advantages of Java 8 Optional?

Optional:
Optional is a final Class introduced as part of Java SE 8. It is defined in java.util package.

It is used to represent optional values that is either exist or not exist. It can contain either one value or zero value. If it contains a value, we can get it. Otherwise, we get nothing.

It is a bounded collection that is it contains at most one element only. It is an alternative to “null” value.

Main Advantage of Optional is:

- It is used to avoid null checks.
- It is used to avoid “NullPointerException”.

What is LRU Cache?

LRU Cache stands for Least Recently Used Cache. The size of the cache is fixed and it supports get() and put() operations. When the cache is full, the put() operation removes the least recently used cache.

How to implement LRU Cache in Java?

The LRU cache can be implemented in Java using two data structures – HashMap and a doubly-linked list to store the data.

Why default methods were needed in java 8?

Simplest answer is to enable the functionality of lambda expression in java. Lambda expression are essentially of type of functional interface. To support lambda expressions seamlessly, all core classes have to be modified. But these core classes like java.util.List are implemented not only in JDK classes, but also in thousands of client code as well. Any incompatible change in core classes will back fire for sure and will not be accepted at all. Default methods break this deadlock and allow adding support for functional interface in core classes.

Double colon (::) operator in Java

The double colon (::) operator, also known as method reference operator

How conflicts are resolved while calling default methods?

So far so good. We have got all basics well. Now move to complicated things. In java, a class can implement N number of interface. Additionally, a interface can also extend another interface as well. An if any default method is declared in two such interfaces which are implemented by single class. then obviously class will get confused which method to call.

Rules for this conflict resolution are as follows:

1) Most preferred are the overridden methods in classes. They will be matched and called if found before matching anything.
2) The method with the same signature in the "most specific default-providing interface" is selected. This means if class Animal implements two interfaces i.e. Moveable and Walkable such that Walkable extends Moveable. Then Walkable is here most specific interface and default method will be chosen from here if method signature is matched.
3) If Moveable and Walkable are independent interfaces then a serious conflict condition happen, and compiler will complain then it is unable to decide. The you have to help compiler by providing extra info that from which interface the default method should be called. e.g.

```
Walkable.super.move();

//or

Moveable.super.move();
```

What is the difference between Predicate and Function?

Both are functional interfaces.
Predicate<T> is single argument function and either it returns true or false.This can be used as the assignment target for a lambda expression or method reference.
Function<T,R> is also single argument function but it returns an Object.Here T denotes type of input to the function and R denotes type of Result.

Are you aware of Date and Time API introduced in Java 8? What the issues with Old Date and time API?

Issues with old Date and TIme API:

Thread Safety: You might be already aware that java.util.Date is mutable and not thread safe. Even java.text.SimpleDateFormat is also not Thread-Safe. New Java 8 date and time APIs are thread safe.
Performance: Java 8 's new APIs are better in performance than old Java APIs.

More Readable: Old APIs such Calendar and Date are poorly designed and hard to understand. Java 8 Date and Time APIs are easy to understand and comply with ISO standards.

Can you provide some APIs of Java 8 Date and TIme?

LocalDate, **LocalTime**, and **LocalDateTime** are the Core API classes for Java 8. As the name suggests, these classes are local to context of observer. It denotes current date and time in context of Observer.

Java 8 stream – if-else logic

The 'if-else' condition can be put as a lambda expression in stream.forEach() function in form of a **Consumer** action.

In given example, we are checking *if* a number id even then print a message, *else* print another message for odd numbers.

```
ArrayList<Integer> numberList = new ArrayList<>(Arrays.asList(1,2,3,4,5,6));
Consumer<Integer> action = i -> {
    if(i % 2 == 0) {
        System.out.println("Even number :: " + i);
    } else {
        System.out.println("Odd  number :: " + i);
    }
};

 numberList.stream().forEach(action);
```

Read file line by line – Java 8 Stream

```
private static void readStreamOfLinesUsingFiles() throws IOException
{
    Stream<String> lines = Files.lines(Paths.get("c:/temp", "data.txt"));

    Optional<String> hasPassword = lines.filter(s ->
s.contains("password")).findFirst();

    if(hasPassword.isPresent())
    {
        System.out.println(hasPassword.get());
    }

    //Close the stream and it's underlying file as well
    lines.close();

}
```

Java Boxed Stream Example

1. IntStream – stream of ints

Example to **convert int stream to List of Integers**.

```
//Get the collection and later convert to stream to process elements
List<Integer> ints = IntStream.of(1,2,3,4,5)
                .boxed()
                .collect(Collectors.toList());
 System.out.println(ints);
 //Stream operations directly
Optional<Integer> max = IntStream.of(1,2,3,4,5)
                .boxed()
                .max(Integer::compareTo);

 System.out.println(max);
```

Program Output:

```
[1, 2, 3, 4, 5]

5
```

2. LongStream – stream of longs

Example to **convert long stream to List of Longs**.

```
List<Long> longs = LongStream.of(1l,2l,3l,4l,5l)
                .boxed()
                .collect(Collectors.toList());

System.out.println(longs);

Output:
 [1, 2, 3, 4, 5]
```

3. DoubleStream – stream of doubles

Example to **convert double stream to List of Doubles**.

```
List<Double> doubles = DoubleStream.of(1d,2d,3d,4d,5d)
                .boxed()
                .collect(Collectors.toList());

System.out.println(doubles);
```

```
Output:


[1.0, 2.0, 3.0, 4.0, 5.0]
...
```

7. Suppliers

The *Supplier* functional interface is yet another *Function* specialization that does not take any arguments. It is typically used for lazy generation of values. For instance, let's define a function that squares a *double* value. It will receive not a value itself, but a *Supplier* of this value:

```
public double squareLazy(Supplier<Double> lazyValue) {
    return Math.pow(lazyValue.get(), 2);
}
```

This allows us to lazily generate the argument for invocation of this function using a *Supplier* implementation. This can be useful if the generation of this argument takes a considerable amount of time. We'll simulate that using Guava's *sleepUninterruptibly* method:

```
Supplier<Double> lazyValue = () -> {
    Uninterruptibles.sleepUninterruptibly(1000, TimeUnit.MILLISECONDS);
    return 9d;
};

Double valueSquared = squareLazy(lazyValue);
```

Another use case for the Supplier is defining a logic for sequence generation. To demonstrate it, let’s use a static *Stream.generate* method to create a *Stream* of Fibonacci numbers:

```
int[] fibs = {0, 1};
Stream<Integer> fibonacci = Stream.generate(() -> {
    int result = fibs[1];
    int fib3 = fibs[0] + fibs[1];
    fibs[0] = fibs[1];
    fibs[1] = fib3;
```

```
6         return result;
7     });
8
```

The function that is passed to the *Stream.generate* method implements the *Supplier* functional interface. Notice that to be useful as a generator, the *Supplier* usually needs some sort of external state. In this case, its state is comprised of two last Fibonacci sequence numbers.

To implement this state, we use an array instead of a couple of variables, because **all external variables used inside the lambda have to be effectively final**.

Other specializations of *Supplier* functional interface include *BooleanSupplier*, *DoubleSupplier*, *LongSupplier* and *IntSupplier*, whose return types are corresponding primitives.

Q1. What Is Nashorn in Java8?

Nashorn is the new Javascript processing engine for the Java platform that shipped with Java 8. Until JDK 7, the Java platform used Mozilla Rhino for the same purpose. as a Javascript processing engine.

Nashorn provides better compliance with the ECMA normalized JavaScript specification and better runtime performance than its predecessor.

Q2. What Is JJS?

In Java 8, *jjs* is the new executable or command line tool used to execute Javascript code at the console.

8. Consumers

As opposed to the *Supplier*, the *Consumer* accepts a generified argument and returns nothing. It is a function that is representing side effects.

For instance, let's greet everybody in a list of names by printing the greeting in the console. The lambda passed to the *List.forEach* method implements the *Consumer* functional interface:

```
List<String> names = Arrays.asList("John", "Freddy", "Samuel");
names.forEach(name -> System.out.println("Hello, " + name));
```

There are also specialized versions of the *Consumer* — *DoubleConsumer*, *IntConsumer* and *LongConsumer* — that receive primitive values as arguments. More interesting is the *BiConsumer* interface. One of its use cases is iterating through the entries of a map:

```
Map<String, Integer> ages = new HashMap<>();
ages.put("John", 25);
ages.put("Freddy", 24);
ages.put("Samuel", 30);

ages.forEach((name, age) -> System.out.println(name + " is " + age + " years old"));
```

Another set of specialized *BiConsumer* versions is comprised of *ObjDoubleConsumer*, *ObjIntConsumer*, and *ObjLongConsumer* which receive two arguments one of which is generified, and another is a primitive type.

9. Predicates

In mathematical logic, a predicate is a function that receives a value and returns a boolean value.

The *Predicate* functional interface is a specialization of a *Function* that receives a generified value and returns a boolean. A typical use case of the *Predicate* lambda is to filter a collection of values:

```
List<String> names = Arrays.asList("Angela", "Aaron", "Bob", "Claire", "David");

List<String> namesWithA = names.stream()
  .filter(name -> name.startsWith("A"))
  .collect(Collectors.toList());
```

In the code above we filter a list using the *Stream* API and keep only names that start with the letter “A”. The filtering logic is encapsulated in the *Predicate* implementation.

As in all previous examples, there are *IntPredicate*, *DoublePredicate* and *LongPredicate* versions of this function that receive primitive values.

10. Operators

Operator interfaces are special cases of a function that receive and return the same value type. The *UnaryOperator* interface receives a single argument. One of its use cases in the Collections API is to replace all values in a list with some computed values of the same type:

```
1   List<String> names = Arrays.asList("bob", "josh", "megan");
2
3   names.replaceAll(name -> name.toUpperCase());
```

The *List.replaceAll* function returns *void*, as it replaces the values in place. To fit the purpose, the lambda used to transform the values of a list has to return the same result type as it receives. This is why the *UnaryOperator* is useful here.

Of course, instead of *name -> name.toUpperCase()*, you can simply use a method reference:

```
1   names.replaceAll(String::toUpperCase);
```

One of the most interesting use cases of a *BinaryOperator* is a reduction operation. Suppose we want to aggregate a collection of integers in a sum of all values. With *Stream* API, we could do this using a collector, but a more generic way to do it is, would be to use the *reduce* method:

```
1   List<Integer> values = Arrays.asList(3, 5, 8, 9, 12);
2
3   int sum = values.stream()
4     .reduce(0, (i1, i2) -> i1 + i2);
```

The *reduce* method receives an initial accumulator value and a *BinaryOperator* function. The arguments of this function are a pair of values of the same type, and a function itself contains a logic for joining them in a single value of

the same type. **Passed function must be associative**, which means that the order of value aggregation does not matter, i.e. the following condition should hold:

```
1   op.apply(a, op.apply(b, c)) == op.apply(op.apply(a, b), c)
```

The associative property of a *BinaryOperator* operator function allows to easily parallelize the reduction process.

Of course, there are also specializations of *UnaryOperator* and *BinaryOperator* that can be used with primitive values,
namely *DoubleUnaryOperator*, *IntUnaryOperator*, *LongUnaryOperator*, *DoubleBinaryOperator*, *IntBinaryOperator*, and *LongBinaryOperator*.

11. Legacy Functional Interfaces

Not all functional interfaces appeared in Java 8. Many interfaces from previous versions of Java conform to the constraints of a *FunctionalInterface* and can be used as lambdas. A prominent example is the *Runnable* and *Callable* interfaces that are used in concurrency APIs. In Java 8 these interfaces are also marked with a *@FunctionalInterface* annotation. This allows us to greatly simplify concurrency c1

2 Thread thread = new Thread(() -> System.out.println("Hello From Another Thread"));

thread.start();ode:

Comparator

Comparator is used when we want to sort a collection of objects which can be compared with each other. This comparison can be done using Comparable interface as well, but it restrict you compare these objects in a single particular way only. If you want to sort this collection, based on multiple criterias/fields, then you have to use Comparator only.

Additionally I have written one method which always return a list of Employees in unsorted order.

```
private static List<Employee> getEmployees(){
    List<Employee> employees  = new ArrayList<>();
    employees.add(new Employee(6,"Yash", "Chopra", 25));
    employees.add(new Employee(2,"Aman", "Sharma", 28));
    employees.add(new Employee(3,"Aakash", "Yaadav", 52));
    employees.add(new Employee(5,"David", "Kameron", 19));
    employees.add(new Employee(4,"James", "Hedge", 72));
    employees.add(new Employee(8,"Balaji", "Subbu", 88));
    employees.add(new Employee(7,"Karan", "Johar", 59));
    employees.add(new Employee(1,"Lokesh", "Gupta", 32));
    employees.add(new Employee(9,"Vishu", "Bissi", 33));
    employees.add(new Employee(10,"Lokesh", "Ramachandran", 60));
    return employees;
}
```

2) Sort by first name

Basic usecase where list of employees will be sorted based on their first name.

```
    List<Employee> employees  = getEmployees();

    //Sort all employees by first name
    employees.sort(Comparator.comparing(e -> e.getFirstName()));

    //OR you can use below
```

```
    employees.sort(Comparator.comparing(Employee::getFirstName));


    //Let's print the sorted list
    System.out.println(employees);
```

Output: //Names are sorted by first name

```
[
    [3,Aakash,Yaadav,52],
    [2,Aman,Sharma,28],
    [8,Balaji,Subbu,88],
    [5,David,Kameron,19],
    [4,James,Hedge,72],
    [7,Karan,Johar,59],
    [1,Lokesh,Gupta,32],
    [10,Lokesh,Ramachandran,60],
    [9,Vishu,Bissi,33],
    [6,Yash,Chopra,25]
]
```

3) Sort by first name – ‘reverse order’

What if we want to sort on first name but in revered order. It’s really very easy; use reversed() method.

```
    List<Employee> employees  = getEmployees();


    //Sort all employees by first name; And then reversed
    Comparator<Employee> comparator = Comparator.comparing(e -> e.getFirstName());
    employees.sort(comparator.reversed());


    //Let's print the sorted list
    System.out.println(employees);
```

Output: //Names are sorted by first name

```
[[6,Yash,Chopra,25],
[9,Vishu,Bissi,33],
[1,Lokesh,Gupta,32],
[10,Lokesh,Ramachandran,60],
[7,Karan,Johar,59],
[4,James,Hedge,72],
[5,David,Kameron,19],
[8,Balaji,Subbu,88],
[2,Aman,Sharma,28],
[3,Aakash,Yaadav,52]]
```

4) Sort by last name

We can use similar code to sort on last name as well.

```
    List<Employee> employees  = getEmployees();

    //Sort all employees by first name
    employees.sort(Comparator.comparing(e -> e.getLastName()));

    //OR you can use below
    employees.sort(Comparator.comparing(Employee::getLastName));

    //Let's print the sorted list
    System.out.println(employees);
```

Output: //Names are sorted by first name

```
[[9,Vishu,Bissi,33],
[6,Yash,Chopra,25],
[1,Lokesh,Gupta,32],
[4,James,Hedge,72],
[7,Karan,Johar,59],
[5,David,Kameron,19],
[10,Lokesh,Ramachandran,60],
```

[2,Aman,Sharma,28],

[8,Balaji,Subbu,88],

[3,Aakash,Yaadav,52]]

5) Sort on multiple fields – thenComparing()

Here we are sorting list of employees first by their first name, and then sort again the list of last name. Just as we apply sorting on SQL statements. It's actually a very good feature.

Now you don't need to always use sorting on multiple fields in SQL select statements, you can sort them in java as well.

```
List<Employee> employees  = getEmployees();


//Sorting on multiple fields; Group by.

Comparator<Employee> groupByComparator =
Comparator.comparing(Employee::getFirstName)

                                    .thenComparing(Employee::getLastName);

employees.sort(groupByComparator);


System.out.println(employees);
```

Output:

[3,Aakash,Yaadav,52],

[2,Aman,Sharma,28],

[8,Balaji,Subbu,88],

[5,David,Kameron,19],

[4,James,Hedge,72],

[7,Karan,Johar,59],

[1,Lokesh,Gupta,32], //These both employees are

[10,Lokesh,Ramachandran,60], //sorted on last name as well

[9,Vishu,Bissi,33],

[6,Yash,Chopra,25]

5) Parallel sort (with multiple threads)

You can sort the collection of objects in parallel using multiple threads as well. It is going to be very fast if collection is big enough having thousands of objects. For small collection of objects, normal sorting is good enough and recommended.

```
//Parallel Sorting

Employee[] employeesArray = employees.toArray(new Employee[employees.size()]);


//Parallel sorting

Arrays.parallelSort(employeesArray, groupByComparator);


System.out.println(employeesArray);
```

Output:

[3,Aakash,Yaadav,52],

[2,Aman,Sharma,28],

[8,Balaji,Subbu,88],

[5,David,Kameron,19],

[4,James,Hedge,72],

[7,Karan,Johar,59],

[1,Lokesh,Gupta,32], //These both employees are

[10,Lokesh,Ramachandran,60], //sorted on last name as well

[9,Vishu,Bissi,33],

[6,Yash,Chopra,25]

That's all for using lambda with Comparator to sort objects. Please share with all of us if you know more techniques around this concept.

```
//Compare by Id

Comparator<Employee> compareById_1 = Comparator.comparing(e -> e.getId());


Comparator<Employee> compareById_2 = (Employee o1, Employee o2) ->
o1.getId().compareTo( o2.getId() );


//Compare by firstname

Comparator<Employee> compareByFirstName = Comparator.comparing(e ->
e.getFirstName());


//how to use comparator

Collections.sort(employees, compareById);
```