JAVA 8 NEW FEATURES

- **Lambda expressions**
- **Functional interfaces**
- **Method references**
- **Default methods**
- **Stream API**
- **Optional**
- **Nashorn, JavaScript Engine**

With the Java 8 release, Java provided supports for **functional programming**, new JavaScript engine, new APIs for date time manipulation, new streaming API, etc.

Lambda expressions

Lambda expressions provide a **clear and concise** way to represent one method interface using an expression.
Interface with only one abstract method is also called as functional interface in Java 8.
A lambda expression is composed of three parts such as Parameter list, Arrow token and Body.
Syntax:

parameter **->** *expression body*

Example:

1. (int x, int y) -> x + y
2. () -> 42
3. (String s) -> { System.out.println(s); }

The body can be either a single expression or a statement block. In the expression form, the body is simply evaluated and returned. In the block form, the body is evaluated like a method body and a return statement returns control to the caller of the anonymous method.

Following are the important characteristics of a lambda expression −

1. **Optional type declaration** − No need to declare the type of parameter. The compiler can inference the same from the value of the parameter.

   (x, y) -> x + y

2. **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.

   String s -> { System.out.println(s); }

3. **Optional curly braces** − No need to use curly braces in expression body if the body contains single statement.

   String s -> System.out.println(s);

4. **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

The body of a lambda expression can be a block statement or a single expression. A block statement is enclosed in braces; a single expression is not enclosed in braces.
When an expression is used as the body, it is evaluated and returned to the caller. The following two lambda expressions are the same; one uses a block statement and the other an expression:

```
// Uses a block statement.
(int x, int y) -> { return x + y; }
// Uses an expression.
(int x, int y) -> x + y;
```

If the expression evaluates to void, nothing is returned to the caller. The following two lambda expressions are the same; one uses a block statement as the body and the other an expression that evaluates to void:

```
// Uses a block statement
(String msg) -> { System.out.println(msg); }
// Uses an expression
(String msg) -> System.out.println(msg);
```

Lambda expressions let us express instances of single-method interfaces (referred to as functional interfaces) more compactly.
Lambda expression facilitates functional programming.
Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single abstract method only.

Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.
Example #1: Runnable lambda

```
public class RunnableLambda {
        public static void main(String[] args) {
                // Anonymous Runnable
                Runnable r1 = new Runnable() {
                        @Override
                        public void run() {
                                System.out.println("*****");
                        }
                };

                // Lambda Runnable
```

```
		Runnable r2 = () -> System.out.println("#########");

		new Thread(r1).start();
		new Thread(r2).start();
	}
}
```

Example #2: Comparator lambda

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
class Customer {
	private int cid;
	private String cname;

	Customer(int cid, String cname) {  this.cid = cid;  this.cname = cname; }
	public Integer getCid() { return cid; }
	public String getCname() { return cname; }
	public String toString() { return cid + " " + cname; }
}

public class ComparatorLambda {
	public static void main(String[] args) {
		List<Customer> custs = new ArrayList<Customer>();
		custs.add(new Customer(2, "abc"));
		custs.add(new Customer(1, "xyz"));
		custs.add(new Customer(3, "pqr"));

		// annonymous comparator
		Collections.sort(custs, new Comparator<Customer>() {
			@Override
			public int compare(Customer c1, Customer c2) {
				return c1.getCid().compareTo(c2.getCid());
			}
		});
		System.out.println(custs.toString());

		// lambda comparator
		//Collections.sort(custs, (Customer c1, Customer c2)-> c1.getCname().compareTo(c2.getCname()));
		Collections.sort(custs, (c1, c2)-> c1.getCname().compareTo(c2.getCname()));
		System.out.println(custs.toString());
```

```
    }
}
```

Lambda expressions also improve the iterate process in collections.
The java.util.Iterable interface has been enhanced in Java 8 with a special method named **forEach**(), which accepts a parameter of type Consumer.
Example:

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;
public class ForEachEx1 {
    public static void main(String[] args) {
        List<String> friends = Arrays.asList("Brian", "Nate", "Neal", "Raju", "Sara", "Scott");

        //Imperative style using Iterator
        Iterator<String> ittr = friends.iterator();
        while(ittr.hasNext()){
            String friend = ittr.next();
            System.out.print(friend +" ");
        }
        System.out.println("");

        //Imperative style using Enhanced for loop
        for(String friend : friends){
            System.out.print(friend +" ");
        }
        System.out.println("");

        //Using forEach() method added to Iterable in Java 8 and anonymous inner class
        friends.forEach(new Consumer<String>() {
            @Override
            public void accept(String friend){
                System.out.print(friend +" ");
            }
        });
        System.out.println("");

        //Using forEach() method added to Iterable and Lambda expression Java 8
        friends.forEach(friend-> System.out.print(friend+" "));
        System.out.println("");
```

```
                //Using forEach() method added to Iterable, Lambda expression and Method reference in Java 8
                friends.forEach(System.out::println);
        }
}
```

Conclusion: Lambda expressions helped us concisely iterate over a collection.

Functional interfaces

The functional interface contains:

1. Exactly one abstract method	[mandatory]
2. Default methods	[optional]
3. Static methods	[optional]
4. Methods inherited from Object class	[optional]

In Java 8, many new functional interfaces such as **java.util.function**.Consumer, Function, Predicate, etc were added to leverage Lambda expressions.

Functional interfaces are used as assignment target for lambda expression or method reference.

The Java compiler will take either a lambda expression or a reference to a method where an implementation of a functional interface is expected.

Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
public class FuncationalInterfaceEx1 {
        public static void main(String[] args) {
                List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

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

                System.out.println("Print all numbers:");
                // pass n as parameter
                eval(list, n -> true);

                // Predicate<Integer> predicate1 = n -> n%2 == 0
                // n is passed as parameter to test method of Predicate interface
                // test() method will return true if n%2 comes to be zero
                System.out.println("Print even numbers:");
                eval(list, n -> n % 2 == 0);

                // Predicate<Integer> predicate2 = n -> n > 3
                // n is passed as parameter to test method of Predicate interface
                // test() method will return true if n is greater than 3.
                System.out.println("Print numbers greater than 3:");
                eval(list, n -> n > 3);
        }

        public static void eval(List<Integer> list, Predicate<Integer> predicate) {
                //Using Imperative way
                /*for (Integer i : list) {
                    if (predicate.test(i)) {
                        System.out.println(i + " ");
                    }
                }*/

                //Using Lambda and forEach() in Java 8
                list.forEach(i -> {
                            if (predicate.test(i)) {
                                System.out.println(i + " ");
                            }
                    }
                );
        }
}
```

Method references

The method reference is shorthand for the lambda expression. By referring to a method name explicitly, our code can gain better readability. When we need a method reference, the target reference is placed before the delimiter **::** and the name of the method is provided after it.

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

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

Using method references make our lambda expressions more readable and concise.
If body in lambda contains single expression then we can use a method reference in place of lambda.

Example #1:

```
import java.util.Arrays;
import java.util.List;
public class MethodRefEx1 {
    public static void main(String[] args) {
        List<String> friends = Arrays.asList("Brian", "Nate", "Neal", "Raju", "Sara", "Scott");
```

```
		//Lambda approach
		friends.forEach(friend-> System.out.println(friend));

		//Method reference approach
		friends.forEach(System.out::println);
	}
}
```

Example #2:

```
import java.util.Arrays;
import java.util.List;
public class MethodRefEx2 {
	public static void main(String[] args) {
		List<Integer> asList = Arrays.asList(4, 6, 10, 3, 6);

		System.out.println("Descending order");
		asList.sort(MethodRefEx2::comapreToStatic);
		asList.forEach(System.out::println);

		MethodRefEx2 methodTest = new MethodRefEx2();
		System.out.println("Ascending order");
		asList.sort(methodTest::comapreToNonStatic);
		asList.forEach(System.out::println);
	}

	private static Integer comapreToStatic(Integer e1, Integer e2) {
		return e2.compareTo(e1);
	}
	private Integer comapreToNonStatic(Integer e1, Integer e2) {
		return e1.compareTo(e2);
	}
}
```

Default methods

Whenever new method is added to an interface then all existing concrete classes need to be modified. This is problematic because the Java 8 API added many new methods in existing interfaces such as sort() method in List interface, forEach() method in Iterable interface, etc.

To avoid this problem, Java 8 allows default methods in interface. **Hence interface can declare methods with implementation code called as default methods**.

Note: Interface in Java 8 allows both default methods and static methods.

Since interfaces can contain concrete method implementation hence subclasses can use default methods.

The sort() method in the List interface is new to Java 8 and is defined as follows:

```
	default void sort(Comparator c){
		Collections.sort(this, c);
	}
```

Example #1:

```
import java.util.*;
class Student {
	private int sno;
	private String sname;
	Student(int sno, String sname) { this.sno = sno; this.sname = sname; }
	public int getSno() { return sno; }
	public String getSname() { return sname; }
	@Override
	public String toString() { return sno + " " + sname; }
}
class SnoSort implements Comparator<Student> {
	@Override
	public int compare(Student s1, Student s2) {
		Integer iRef1 = s1.getSno();
		Integer iRef2 = s2.getSno();
		return iRef1.compareTo(iRef2);
	}
}
public class DefaultMethodEx1 {
	public static void main(String[] args) {
		List<Student> students = new Vector<Student>();
		students.add(new Student(2, "abc"));
		students.add(new Student(1, "xyz"));
		students.add(new Student(3, "pqr"));
		System.out.println(students.toString());

		//Using sort() in Collections class before Java 8
		/*Collections.sort(students, new SnoSort());
		System.out.println(students.toString());*/

		//Using sort() in List interface in Java 8
		students.sort(new SnoSort());
		System.out.println(students.toString());
	}
}
```

The forEach() method in the Iterable interface is new to Java 8 and is defined as follows:

```
	default void forEach(Consumer){
		…
	}
```

Example #2:

```
import java.util.*;
public class DefaultMethodEx2 {
	public static void main(String[] args) {
		List<String> friends = Arrays.asList("Brian", "Nate", "Neal", "Raju", "Sara", "Scott");

		// Old approach using Iterator
		Iterator<String> ittr = friends.iterator();
		while (ittr.hasNext()) {
			String friend = ittr.next();
			System.out.print(friend + " ");
		}
		System.out.println("");

		//Using forEach() default method in Java 8
		friends.forEach(friend->System.out.print(friend+" "));
	}
}
```

This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8. For example, ‘List’ or ‘Collection’ interfaces do not have ‘forEach’ method declaration. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not to implement the same.

Example #3:

```
interface Vehicle{
	default public void disp(){
		System.out.println("I am vehicle");
	}
}
class Car implements Vehicle{
	/*public void disp(){
		System.out.println("I am car");
	}*/
}
```

```
public class DefaultMethodsEx3 {
	public static void main(String[] args) {
		Car obj = new Car();
		obj.disp();
	}
}
```

O/P:

```
I am vehicle
```

The main users of default methods are library designers. The default methods were introduced to evolve libraries such as the Java API in a compatible way. The default methods let library designers evolve APIs **without breaking** existing code because classes implementing an updated interface automatically inherit a default implementation.

Abstract classes vs. interfaces in Java 8

From Java 8, both abstract class and interface contain abstract methods and concrete methods.
The following are the differences:

1) A class can extend only one abstract class, but a class can implement multiple interfaces.
2) An abstract class can enforce a common state through instance variables (fields). An interface can't have instance variables.

Resolution Rules

With the introduction of default methods in Java 8, there's a possibility of a class inheriting more than one method with the same signature, which causes method conflicts.
To avoid method conflict, we must override the method in class and then in its body explicitly call the method we wish to use.

Example #4:

```
interface A{
	default void hello(){
		System.out.println("Hello from A");
	}
}
interface B{
	default void hello(){
		System.out.println("Hello from B");
	}
}
class C implements A, B{
	public void hello(){
		B.super.hello();
	}
}
public class DefaultMethodEx4 {
        public static void main(String[] args) {
                new C().hello();
        }
}
```

O/P:

```
Hello from B
```

Stream API

Using stream, we can **process data in a declarative way** similar to SQL statements. For example, consider the following SQL statement.

    select * from employee where sal= (select max(sal) from employee);

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer has to use loops and make repeated checks.

To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively.

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

default public Stream **stream**() − Returns a sequential stream considering collection as its source.

default public Stream **parallelStream**() − Returns a parallel Stream considering collection as its source.

The **Stream** interface contains following methods:

1) forEach(Consumer)
2) map(Function)
   This method transforms all elements in collection from one form to another form.
3) Reduce()
4) filter(Predicate)
   Returns new stream.
   …

Using the methods of this interface, we can compose a sequence of calls so that the code reads and flows in the same way we'd state problems, making it easier to read.

Example #1:

```
import java.util.Arrays;
import java.util.List;
public class StreamEx1 {
        public static void main(String[] args) {
                List<Integer> values = Arrays.asList(1, 2, 3, 4, 5, 6);

                //without streams
                int total = 0;
                for (int e : values) {
                        total += e * 2;
                }
                System.out.println(total);

                // Using Stream API in Java 8
                Integer result = values.stream()
                                    .map(e -> e * 2)
                                    .reduce(0, (c, e) -> c + e);
                System.out.println(result);
        }
}
```

Example: **Capitalize each name**

```
import java.util.*;
public class StreamEx2 {
    public static void main(String[] args) {
        List<String> friends = Arrays.asList("Brian", "Nate", "Neal", "Raju", "Sara", "Scott");
        List<String> uppercaseNames = new ArrayList<String>();

        //Imperative style -- older approach
        for (String name : friends) {
            uppercaseNames.add(name.toUpperCase());
        }
        System.out.println(uppercaseNames);
        uppercaseNames.clear();

        //Using Lambda, forEach() in Java 8
        friends.forEach(name->uppercaseNames.add(name.toUpperCase()));
        System.out.println(uppercaseNames);
        uppercaseNames.clear();

        //Using Lambda, forEach(), map() method in Stream interface in Java 8
        friends.stream()
        .map(name->name.toUpperCase())
        .forEach(name-> uppercaseNames.add(name));
        System.out.println(uppercaseNames);
        uppercaseNames.clear();

        //Using Lambda, forEach(), map() method in Stream interface, Method reference in Java 8
        friends.stream()
        .map(String::toUpperCase)
        .forEach(name-> uppercaseNames.add(name));
        System.out.println(uppercaseNames);
        }
}
```

Example: **Filter names start with 'N'**

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class StreamEx3 {
        public static void main(String[] args) {
                List<String> friends = Arrays.asList("Brian", "Nate", "Neal", "Raju", "Sara", "Scott");
                List<String> startsWithN = new ArrayList<String>();

                // Imperative style -- older approach
                for (String name : friends) {
                        if (name.startsWith("N")) {
                                startsWithN.add(name);
                        }
                }
                System.out.println(startsWithN);
                startsWithN.clear();

                // Using stream(), filter() and Lambda in Java 8
                Stream<String> newStream = friends.stream().filter(name-> name.startsWith("N"));
                startsWithN = newStream.collect(Collectors.toList());

                //startsWithN = friends.stream().filter(name->name.startsWith("N")).collect(Collectors.toList());

                System.out.println(startsWithN);
        }
}
```

Optional

Emphasis on best practices to handle null values properly.

Example :

```
import java.util.Optional;

public class OptionalEx1 {
        public static void main(String[] args) {
                Integer value1 = null;
                Integer value2 = new Integer(10);
```

```
                    //System.out.println(value1 + value2); //null pointer exception

                    int sum = Optional.ofNullable(value1).orElse(new Integer(0)) + Optional.ofNullable(value2).orElse(new Integer(0));
                    System.out.println("The result:"+sum); //10
          }
}
```

What is Lambda?
Ans) Lambda is a functional programming used to provide inline implementation of a functional interface.

What is functional interface?
Ans) The interface with only one method.

What is the main advantages of Lambda?
Ans) Lambda expressions provide a **clear and concise** way to implement functional interfaces.