Exploring the Evolution of Functional Programming in Java: A Comparison of Java 8 to Earlier Versions

Kartik M. Soni, ksoni@ncsu.edu

Functional programming has become an increasingly popular paradigm in software development, and Java has evolved over the years to incorporate more functional programming features. In this paper, I will explore the evolution of functional programming in Java and compare the functional programming capabilities of Java 8 to earlier versions.

After researching the documentation available online, it has become apparent that Java 8 introduced several interesting features. While I had previously used some of these features, I was unaware that they were specific to Java 8. Among the most significant features are functional programming, lambda expressions, and the stream API. With these additions, the emphasis shifts from objects to functions, which define the behavior of objects. One particularly important capability is the ability to define methods within interfaces - which adds an additional level of flexibility to writing code in Java.

Here are some of them:

1. Interfaces

As we know, interface methods are by default public and abstract. (In abstract class, we can also define methods, whereas all methods of an interface are abstract). Java 8 has added the capability of adding default and static methods to interfaces.

This was done because, let's say you have an interface, and a lot of different classes implement it. If you update that interface with another abstract method, you will need to implement that method in all classes that implement the interface, or else your code will fail. Thus, to avoid issues like these, Java 8 introduced the ability of implementing default methods in interfaces.

Further, multiple inheritance is not possible with classes (due to the diamond problem), but before Java 8, a class could implement multiple interfaces since the methods were abstract. But multiple inheritance with interfaces is not supported with Java 8 in a straightforward manner since the methods are not necessarily abstract. So let's say, you have an interface A with a method definition test(){}, and an interface B with another definition of test(){}. In that case, if a class C implements A,B; then the class C would need to define the method test(){} itself so that there is no confusion. Further, if a class extends another class *and* implements an interface and

if there's a conflict in method name between the parent class and the interface, priority will be given to the parent class methods and not the default methods of the interface.

Lastly, Java 8 also supports static methods in interfaces. Here's an example of this change:

```
public interface Vehicle {

    void start();
    void stop();

    default void accelerate() {
        System.out.println("Vehicle is accelerating");
    }

    static void printVehicleType() {
        System.out.println("This is a Vehicle interface");
    }
}

public class Car implements Vehicle {

    @Override
    public void start() {
        System.out.println("Car is starting");
    }

    @Override
    public void stop() {
        System.out.println("Car is stopping");
    }

    // No implementation needed for accelerate()

}

public class Main {

    public static void main(String[] args) {

        Vehicle car = new Car();
        car.start();
        car.accelerate();
```

```
        car.stop();

        Vehicle.printVehicleType();

    }

}
```

This code gives the following output:

```
Car is starting
Vehicle is accelerating
Car is stopping
This is a Vehicle interface
```

2. Lambda Expressions

Lambda expressions in a nutshell are a way of creating anonymous classes of functional interfaces easily (interfaces with exactly 1 abstract method). Just like JavaScript has anonymous functions and Python has lambda functions, Java 8 introduced lambda expressions. These allow us to pass around blocks of code as values - which is really cool if you want to make your code more concise, readable, and modular.

Let's say we have an interface:

```
interface Vehicle
{
    void drive(String name);
}
```

This is how we can use it in a main method - instead of explicitly implementing the interface with a class, we do it directly using an anonymous class:

```
public class Test{
    public static void main(String[] args){
        Vehicle car;
        car = new Vehicle()
        {
            public void drive(String name)
            {
                System.out.println("Weee goes " + name);
            }
```

```
        };
        car.drive();
    }
}
```

Now, Lambda enables us to do this:

```
public class Test{
    public static void main(String[] args){
        Vehicle car;
        car = (name) -> System.out.println("Weee goes " + name);
        car.drive("Tesla");
    }
}
```

The lambda expression can only be used for functional interfaces - with a single abstract method like the Vehicle interface. If the interface had multiple abstract methods, then it would not be a functional interface and lambda could not be used to implement it. In that case, we would need to revert back to the anonymous class approach.

3. ForEach Method

Up until Java 8, we used to work with “External Loops” - fetching data from outside the collection.

Take this example of printing values from a list:

```
List<Integer> myArr = Arrays.asList(1,2,3,4,5);
//For loop
for(int i=0; i<myArr.size(); i++){
    System.out.println(myArr.get(i));
}

//Enhanced For loop
for(int i : myArr){
    System.out.println(i);
}
```

Both of these are external loops. With Java 8, we have internal loops:

```
myArr.forEach(i -> System.out.println(i));
```

forEach is an internal part of Collections - thus, it will be much faster than external for loops. This is beneficial when working with large amounts of data. Note that this is a lambda expression which implements the consumer interface. forEach method takes a parameter which is an object of Consumer - which in turn is a functional interface. The values in myArr are passed one by one to the interface. Thus, *i is the argument of the only method in the interface which accepts it and prints it.*

For further clarification, this is part of what's happening behind the scenes:

```
Consumer obj = i -> System.out.println(i);
myArr.forEach(obj);
```

4. Stream API

Stream in Java 8 enables us to execute methods like filter, map, and reduce on a stream of elements that can be generated from a collection. When working with a lot of data, the filter method can make it easier to filter elements of a collection out based on some condition.

Example usage:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> evenNumbers = numbers.stream()
                        .filter(x -> x % 2 == 0)
                        .collect();

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

- The stream API is used to create a stream of numbers. With stream(), we have another method, parallelStream() - which automatically creates multiple threads - based on the number of cores available.
- Stream also has a lot of other methods that help in processing a large amount of data - like filter(), map(), and more.
- Once you use a stream's values, they cannot be reused.
- Stream has 2 kinds of methods: Intermediate and Terminate.
- Since filter() is an intermediate method, it will not give any output. Thus, we use terminate methods alongside it.
- Stream API also provides us with Optional in case our list is empty

5. Date and Time API

Previously, the Date classes existed in java.sql and java.util. Also, they were not thread safe.

- We have a special package for it, java.time (java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime)
- Immutable, just like strings

This new API is designed to be more intuitive and less error-prone than the previous date and time API, and provides a range of new features such as support for time zones, better handling of daylight saving time, and more.

6. Method reference

In a nutshell, method reference allows you to refer to a method of a class or an object without invoking it. It's an even shorter way of writing a lambda expression. Java8 introduced the concept of passing a function to a function - that is, higher order functions (or “Call by Method”). To pass a method, we need to specify two colons before it, which in turn need to be preceded by the class to which the method belongs to.

For example:

```
List<Integer> nums = Arrays.asList(1,2,3,4,5);
nums.forEach(System.out::println);
```

Another example:

```
interface Parser{
 public String parse(String s);
}

class StringConverter{
 public static String convert(String s){
  return s+" goes brrr";
 }
}

public class Demo {
 public static void main(String[] args) {
  Parser p = StringConverter::convert;
  System.out.println(p.parse("Kartik"));
 }
```

```
}
```

Here, the method reference StringConverter::convert is used to

- Create an instance of the Parser interface
- Implement the parse method of the Parser interface

When the parse method of the p object is called, the convert method of the StringConverter class is invoked with the provided argument, and its result is returned.

7. Miscellaneous

- Java 8 introduced a new type of array - the **parallel array**, which is more efficient when processing large amounts of data in parallel.
- Java 8 also added **Improved type inference** - which makes it easier to write generic code without having to specify the types explicitly.

8. Example functional Java API using these features

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

public class EmployeeManagementSystem {
   private List<Employee> employees;

   public EmployeeManagementSystem() {
       employees = new ArrayList<>();
   }

   public void addEmployee(Employee employee) {
       employees.add(employee);
   }

   public List<Employee> getEmployees() {
       return employees;
   }

   public List<Employee> getEmployeesByDepartment(String department) {
       return employees.stream()
               .filter(e -> e.getDepartment().equals(department))
```

```
                .toList();
    }

    public double getAverageSalary() {
        return employees.stream()
                .mapToDouble(Employee::getSalary)
                .average()
                .orElse(0);
    }

    public Employee getEmployeeWithMaxSalary() {
        return employees.stream()
                .max((e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary()))
                .orElse(null);
    }

    public List<Employee> getEmployeesByAge(int age) {
        LocalDate today = LocalDate.now();
        LocalDate birthDate = today.minusYears(age);

        return employees.stream()
                .filter(e -> e.getBirthDate().isBefore(birthDate))
                .toList();
    }

    public void giveRaise(double percentage) {
        employees.forEach(e -> e.setSalary(e.getSalary() * (1 + percentage / 100)));
    }
}

class Employee {
    private String name;
    private String department;
    private LocalDate birthDate;
    private double salary;

    public Employee(String name, String department, LocalDate birthDate, double salary)
{
        this.name = name;
        this.department = department;
        this.birthDate = birthDate;
        this.salary = salary;
```

```
    }

    public String getName() {
        return name;
    }

    public String getDepartment() {
        return department;
    }

    public LocalDate getBirthDate() {
        return birthDate;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

Now, here's how a Java API with the exact same functionality would look before Java 8:

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

public class EmployeeService {
    private List<Employee> employees;

    public EmployeeService() {
        employees = new ArrayList<Employee>();
    }

    public void addEmployee(Employee employee) {
        employees.add(employee);
```

```
}

public List<Employee> getEmployees() {
    return employees;
}

public List<Employee> getEmployeesByDepartment(String department) {
    List<Employee> filteredEmployees = new ArrayList<Employee>();
    for (Employee employee : employees) {
        if (employee.getDepartment().equals(department)) {
            filteredEmployees.add(employee);
        }
    }
    return filteredEmployees;
}

public double getAverageSalary() {
    if (employees.isEmpty()) {
        return 0;
    }
    double sum = 0;
    for (Employee employee : employees) {
        sum += employee.getSalary();
    }
    return sum / employees.size();
}

public Employee getEmployeeWithMaxSalary() {
    if (employees.isEmpty()) {
        return null;
    }
    Employee maxEmployee = employees.get(0);
    for (int i = 1; i < employees.size(); i++) {
        Employee employee = employees.get(i);
        if (employee.getSalary() > maxEmployee.getSalary()) {
            maxEmployee = employee;
        }
    }
    return maxEmployee;
}

public List<Employee> getEmployeesByAge(int age) {
```

```
        List<Employee> filteredEmployees = new ArrayList<Employee>();
        for (Employee employee : employees) {
            if (employee.getBirthDate().getYear() <= (LocalDate.now().getYear() - age))
{
                filteredEmployees.add(employee);
            }
        }
        return filteredEmployees;
    }

    public void giveRaise(double percentage) {
        for (Employee employee : employees) {
            double newSalary = employee.getSalary() * (1 + percentage / 100);
            employee.setSalary(newSalary);
        }
    }
}

class Employee {
    private String name;
    private String department;
    private LocalDate birthDate;
    private double salary;

    public Employee(String name, String department, LocalDate birthDate, double salary)
{
        this.name = name;
        this.department = department;
        this.birthDate = birthDate;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public String getDepartment() {
        return department;
    }

    public LocalDate getBirthDate() {
        return birthDate;
```

```
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

Clearly, the code is more verbose and requires more manual iteration over collections.

Now, here are some unit tests for the original Java8 code using JUnit:

```
import org.junit.Test;
import java.time.LocalDate;
import java.util.List;
import static org.junit.Assert.*;

public class EmployeeManagementSystemTest {

    @Test
    public void testAddEmployee() {
        EmployeeManagementSystem ems = new EmployeeManagementSystem();
        Employee employee = new Employee("Kartik Soni", "Software",
LocalDate.parse("2000-09-22"), 50000.00);
        ems.addEmployee(employee);
        assertEquals(1, ems.getEmployees().size());
    }

    @Test
    public void testGetEmployeesByDepartment() {
        EmployeeManagementSystem ems = new EmployeeManagementSystem();
        ems.addEmployee(new Employee("Kartik Soni", "Software",
LocalDate.parse("2000-09-22"), 50000.00));
        ems.addEmployee(new Employee("Jane Smith", "Marketing",
LocalDate.parse("1990-01-01"), 60000.00));
        List<Employee> employees = ems.getEmployeesByDepartment("Software");
        assertEquals(1, employees.size());
        assertEquals("Kartik Soni", employees.get(0).getName());
```

```
    }

    @Test
    public void testGetAverageSalary() {
        EmployeeManagementSystem ems = new EmployeeManagementSystem();
        ems.addEmployee(new Employee("Kartik Soni", "Software",
LocalDate.parse("2000-09-22"), 50000.00));
        ems.addEmployee(new Employee("Jane Smith", "Marketing",
LocalDate.parse("1990-01-01"), 60000.00));
        double averageSalary = ems.getAverageSalary();
        assertEquals(55000.00, averageSalary, 0.001);
    }

    @Test
    public void testGetEmployeeWithMaxSalary() {
        EmployeeManagementSystem ems = new EmployeeManagementSystem();
        ems.addEmployee(new Employee("Kartik Soni", "Software",
LocalDate.parse("2000-09-22"), 50000.00));
        ems.addEmployee(new Employee("Jane Smith", "Marketing",
LocalDate.parse("1990-01-01"), 60000.00));
        Employee maxSalaryEmployee = ems.getEmployeeWithMaxSalary();
        assertEquals("Jane Smith", maxSalaryEmployee.getName());
    }

    @Test
    public void testGetEmployeesByAge() {
        EmployeeManagementSystem ems = new EmployeeManagementSystem();
        ems.addEmployee(new Employee("Kartik Soni", "Software",
LocalDate.parse("2000-09-22"), 50000.00));
        ems.addEmployee(new Employee("Jane Smith", "Marketing",
LocalDate.parse("1990-01-01"), 60000.00));
        List<Employee> employees = ems.getEmployeesByAge(33);
        assertEquals(1, employees.size());
        assertEquals("Jane Smith", employees.get(0).getName());
    }

    @Test
    public void testGiveRaise() {
        EmployeeManagementSystem ems = new EmployeeManagementSystem();
        ems.addEmployee(new Employee("Kartik Soni", "Software",
LocalDate.parse("2000-09-22"), 50000.00));
```

```
        ems.addEmployee(new Employee("Jane Smith", "Marketing",
LocalDate.parse("1990-01-01"), 60000.00));
        ems.giveRaise(10.00);
        assertEquals(55000.00, ems.getEmployees().get(0).getSalary(), 0.001);
        assertEquals(66000.00, ems.getEmployees().get(1).getSalary(), 0.001);
    }
}
```