Fw: USAA - Java Features Training - SharePoint Notes

Oviya P <oviya.p@hcltech.com>
Mon 7/29/2024 2:28 PM
To:Oviya P <oviya.p@hcltech.com>

Subject: RE: USAA - Java Features Training - SharePoint Notes

Day wise hands-on shared for completion, Please complete and share the task files to your RM.

Training Hands-On Examples Shared:

Day 1: Hands-On:

1. Implement 1 functional Interface and implement the method with using lambda expression and without lambda expression
2. Implement Employee Object with Map() and Filter() stream API methods.
3. Implement any 2 Predefined functional interfaces and understand how to implement with and without lambda expression.

Day 2: Hands-On:

1. Java static method & Default method implementation. Use Book Object for reference.
2. Implement method reference using Custom Math class (Arithmetic operations) and perform operations using method reference. (Add, Sub, Division) – Method static ref, Non Static (Instance reference), Constructor reference
3. Create an Employee Object and store the object values in a list with some empty and proper employee values (EmpId, EmpName, EmpRole) then check the value is null or valid and print else use custom exception if not found (Optional methods, IsPresent, Nullable, OrElse, OrElseThrow)

Day-3 Hands-On (Case Study)

Objective:
The primary goal is to use Java 8 stream and collection functionalities to efficiently perform various operations on the employee dataset, such as filtering, mapping, sorting, grouping, and summarizing.

Steps:-

1. Data Preparation:
- Objective: Understand the basics of class definition and data initialization in Java.
- Details:
  - Define the `Employee` class with attributes (`name`, `department`, `salary`, `yearsOfService`).
  - Encapsulate fields using private access modifiers and provide public getter methods.
  - Create a list of `Employee` objects to simulate a dataset for subsequent operations.
  - Use constructors for initializing objects efficiently.

2. Filtering and Mapping:
- Objective: Learn how to filter and transform data using streams.
- Details:
  - Apply the concept of streams with `stream()` method on the list of employees .
  - Use `filter()` to selectively include or exclude elements based on specified conditions (e.g., years of service or department)..
  - Use `map()` to transform each element of the stream.
  - Learn the benefits of immutability in stream operations.

3. Sorting:
- Objective: Explore sorting techniques using Java 8 streams.
- Details:
  - Apply the `sorted()` method to order elements within the stream.
  - Use `Comparator` for custom sorting criteria (e.g., by name, salary, or years of service)..
  - Do sorting in both ascending and descending order.
  - Discuss the difference between natural ordering and custom comparator-based ordering.

4. Grouping:
- Objective: Understand how to group data by specific attributes.
- Details:
  - Apply `groupingBy()` to categorize elements based on a chosen attribute.
  - Group employees by department to create a map with departments as keys and lists of employees as values.
  - Learn the flexibility of grouping for various use cases.

5. Summarization:
- Objective: Learn about summarization using collectors in streams.
- Details:
  - Utilize collectors like `summarizingInt()` to gather statistics on specific attributes.
  - Explore other summarization techniques like `counting()`.
  - Learn the convenience and power of collectors for summarizing data in a concise manner.

6. Integration with Collections:
- Objective: Integrate stream operations with traditional collections.
- Details:
  - Convert streams back to collections using the `collect()` method.
  - List down the scenarios where integration with existing collections is advantageous.
  - Check the seamless compatibility between streams and collections.

Day-4 Hands-On:

1. Implement Before and After switch case statement of Java 17 feature (Create employee Object with Name and role and implement switch case to fetch the department.
2. Implement “Pattern Match for Instance of” using before java 17 with typecast , Without typecasting the Object, Applying some conditional check while doing pattern check for InstanceOf.
3. Implement Record class and Regular class for AnimalsData (Pojo with record and without using record with Main class calls).

Regards,
Oviya Prakasam

Subject: RE: USAA - Java Features Training - Day 4 Notes

Classification: **Internal**

Hi Team,

Please find Day-4 Notes:

Hands-On:

1. Implement Before and After switch case statement of Java 17 feature (Create employee Object with Name and role and implement switch case to fetch the department.
2. Implement “Pattern Match for Instance of” using before java 17 with typecast , Without typecasting the Object, Applying some conditional check while doing pattern check for InstanceOf.
3. Implement Record class and Regular class for AnimalsData (Pojo with record and without using record with Main class calls).

Notes:

Java 17 Feature:

- Java 17 is an LTS (Long-term support) release of the Java programming language.
- As it is an LTS version, it will receive prolonged support and security updates for a minimum of eight years.
- Java 17 comes with various new features that is inclined to improve developer productivity and program efficiency.

Pattern Matching For Switch Statements

- New feature introduce in java 1 called as Pattern Matching for switch Statements
- Developers can make use of this feature to simplify the code of a switch case.
- provides more readable format to write and handle the multiple switch case statements.

Before the feature:

- *Switch statements could only compare the value of a single variable against a series of constants or expressions*
- *previous version of switch statement was limited to byte, short, char, int, Byte, Character, Short, Integer, String, and Enum.*
- *New features allows developer to match against the value of an Object as an type as well*

Traditional Switch Statement

```
public static String getDayOfWeek(int dayNum) {
  String day;
  switch (dayNum) {
    case 1:
      day = "Monday";
      break;
    case 2:
      day = "Tuesday";
      break;
    case 3:
      day = "Wednesday";
      break;
    case 4:
      day = "Thursday";
      break;
    case 5:
      day = "Friday";
      break;
    case 6:
      day = "Saturday";
      break;
    case 7:
      day = "Sunday";
      break;
    default:
      day = "Invalid day";
  }
  return day;
}
```

Switch Statements Since Java 12

```
public static String getDayOfWeek(int dayNum) {
  return switch (dayNum) {
    case 1 -> "Monday";
    case 2 -> "Tuesday";
    case 3 -> "Wednesday";
    case 4 -> "Thursday";
    case 5 -> "Friday";
    case 6 -> "Saturday";
```

```
        case 7 -> "Sunday";
        default -> "Invalid day";
    };
}
```

In Java 12 feature switch case implementation is customized with the case statement "->" with arrow denotation.

This features also eliminated the break statement at each case.

Switch Statements Since Java 17

- Java 17 also allows developers to use variable patterns and type patterns.

```
public static int getLength(Object obj) {
    return switch (obj) {
        case String s -> s.length(); // variable pattern
        case List list && !list.isEmpty() -> list.size(); // type pattern
        case null -> 0; // Allows null checks
        default -> -1;
    };
}
```

Overall feature example in below example: (Switch Case Statements)

```
// Size.java
public enum Size {
   SMALL, MEDIUM, LARGE;
}

// Point.java
public record Point(int i, int j) {
   // Constructor, accessor methods, equals, hashCode, toString are automatically generated
}

// Employee.java
public class Employee {
   private String name;
   private int age;

   public Employee(String name, int age) {
this.name = name;
      this.age = age;
   }

   public String getName() {
      return name;
   }

   // Other getter/setter methods and other functionalities
}

// Main.java
public class Main {
   public static void main(String[] args) {
      Main main = new Main();

      // Testing different types of objects
      System.out.println(main.testType(null));  // NULL value
      System.out.println(main.testType("Hello"));  // It's a String
      System.out.println(main.testType(Size.SMALL));  // Enum, Type
      System.out.println(main.testType(new Point(1, 2)));  // Records Type
      System.out.println(main.testType(new Employee("John", 30)));  // Custom Object's Type
      System.out.println(main.testType(new int[]{1, 2, 3}));  // Array Type
      System.out.println(main.testType(new Employee("John", 30)));  // Conditional Statement
   }

   public String testType(Object obj) {
      return switch (obj) {
         case null -> "NULL value";
         case String str -> "It's a String";
         case Size s -> "Enum, Type";
         case Point p -> "Records Type";
         case Employee e -> "Custom Object's Type";
         case int[] ai -> "Array Type";
         case Employee e && e.getName().equals("John") -> "Conditional Statement";
         default -> "Unknown";
      };
   }
}
```

Pattern Matching For InstanceOf

Description:
Pattern matching for instanceof, introduced as a preview feature in Java 15, simplifies the common pattern of casting an object to a specific type after checking its type with instanceof. It enhances readability and reduces boilerplate code.

Usage:
Pattern matching for instanceof is useful when you need to perform type checks and casts on objects. It allows you to combine the instanceof check and cast operation into a single expression, making code more concise and expressive

Before Java 14:
java
Copy code
```
public class Main {
    public static void main(String[] args) {
        Object obj = "Hello";

        if (obj instanceof String) {
            String str = (String) obj;
            System.out.println("String length: " + str.length());
        } else if (obj instanceof Integer) {
            Integer num = (Integer) obj;
            System.out.println("Integer value: " + num);
        }
    }
}
```
After Java 14:
java
Copy code
```
public class Main {
    public static void main(String[] args) {
        Object obj = "Hello";

        if (obj instanceof String str) {
            System.out.println("String length: " + str.length());
        } else if (obj instanceof Integer num) {
            System.out.println("Integer value: " + num);
        }
    }
}
```

Logical Conditional Checking for Employee Object:
java
Copy code
```
// Employee.java
public class Employee {
    private String name;
    private int age;

    public Employee(String name, int age) {
this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Employee emp1 = new Employee("John", 30);
        Employee emp2 = new Employee("Alice", 25);

        if (isSeniorEmployee(emp1)) {
            System.out.println(emp1.getName() + " is a senior employee.");
        } else {
            System.out.println(emp1.getName() + " is not a senior employee.");
        }

        if (isSeniorEmployee(emp2)) {
            System.out.println(emp2.getName() + " is a senior employee.");
        } else {
            System.out.println(emp2.getName() + " is not a senior employee.");
        }
    }

    public static boolean isSeniorEmployee(Employee emp) {
        return emp instanceof Employee e && e.getAge() > 25;
    }
}
```
In the example:

The Main class demonstrates pattern matching for instanceof before and after Java 14. Before Java 14, explicit casting is required within each instanceof block, while after Java 14, pattern variables (str and num) can be declared directly within the instanceof condition.

Additionally, the Main class showcases a logical conditional check for the Employee object. The isSeniorEmployee method checks if the given Employee object is senior based on a condition (age > 25). This is done using pattern matching for instanceof within the method.

Record Classes: (Important feature of java 14 (preview) and 16 (Standard))

Java Records:

Description:
Java Records, introduced in Java 14 and made standard in Java 16, provide a compact way to declare classes that are transparent holders for immutable data. They are used to define simple data carrier classes with concise syntax, automatically generating boilerplate code such as constructors, accessors, equals(), hashCode(), and toString() methods.

Usage:
Records are used when you need to create data classes that primarily serve to hold immutable data. They are particularly useful for defining DTOs (Data Transfer Objects), POJOs (Plain Old Java Objects), or any other classes where the primary purpose is to encapsulate data rather than behavior. By using records, you can reduce boilerplate code and improve code readability.

Rules for Records:

Records are implicitly final and cannot be extended.
All record components are implicitly final and immutable.
Records automatically generate a constructor that initializes all components.
Records automatically generate accessors (getter methods) for all components.
Records automatically generate implementations for equals(), hashCode(), and toString() methods based on their components.
You can declare additional methods in records, but they cannot override methods from java.lang.Object.

```
// Employee Class (Normal Pojo – Before Record Class)
package com.hcl.test;

import java.util.Objects;

public class Employee {

        private String name;
        private String role;

        public Employee() {
                super();
        }

        public Employee(String name, String role) {
                super();
                this.name = name;
                this.role = role;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getRole() {
                return role;
        }

        public void setRole(String role) {
                this.role = role;
        }

        @Override
        public int hashCode() {
                return Objects.hash(name, role);
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                Employee other = (Employee) obj;
                return Objects.equals(name, other.name) && Objects.equals(role, other.role);
        }
        /*
        * @Override public String toString() { return "Employee [name=" + name +
        * ", role=" + role + "]"; }
        */

}
```

Record Class:

```
package com.hcl.test;

public record EmployeeRecord(String name, String Role){

}
```

Main Class:

```
package com.hcl.test;

public class EmpMain {

	public static void main(String[] args) {

		//Before Record Classes
		Employee emp = new Employee("Oviya", "Software Engineer");
		System.out.println(emp);
		System.out.println(emp.getName());


		//After Record class
		EmployeeRecord empRecord = new EmployeeRecord("userA", "UserRole");
		System.out.println(empRecord);

		System.out.println(empRecord.name());
	}

}
```

Sealed Classes & Sealed interfaces can be covered tomorrow from Java 17:

Regards,
Oviya Prakasam

Subject: RE: USAA - Java Features Training - Day 3 Notes

Classification: **Internal**

Hi Team,

Please find Day3 Notes:

Hands-On:

1. Please find attached file for the Hands-on Reference.

Streams API
- contains collection of objects from some sources(Array,List,Set) and process them sequentially
- Collection framework also stores collection of objects and we can manipulate the object, but streams are only used for processing the data and we cant manipulate the data
- present in java.util.stream.* package
- 2 types of operation

1. Intermediate operation - return stream itself - optional operation - we can chain multiple intermediate operation
 - filter(), map(), flatMap(), sorted(), peek(), distinct(), limit(), skip()

2. Terminal operation - It will traverse into the newly generated stream and return single value - mandatory operation - we cant chain terminal operation
 - toArray(), forEach(), count(), min(), max(), reduce(), collect(),anyMatch(), allMatch(), noneMatch(), findFirst()

1. Creation of Stream - Finite stream - 2 ways
1. stream() - used to generate a stream from some source (ie) array or list or set

String s[]=new String[]{"one","two","three"};
Stream<String> s1=s.stream();

List<String> l1=new ArrayList<>();
l1.add("Ram");
l1.add("Sam");
l1.add("Raj");
Stream<String> s2=l1.stream();

2. of() - used to create our own stream

Stream<Integer> st= Stream.of(1,2,3,4,5);

filter() - intermediate - used to filter the data based on some condition

reduce() - terminal - used to group the data into single value
 - Optional reduce(BinaryOperator)

- Integer reduce(int initialvalue, BinaryOperator)
- Stream reduce(int initialvalue, BinaryOperator b1, BinaryOperator b2)

```
public class Main {
            public static void main(String[] args) {
              List<String> l1=new ArrayList<>();
              l1.add("Ram");
              l1.add("Sam");
              l1.add("Raj");
              l1.add("Tam");
              l1.add("Tim");
              l1.add("Ram");
              l1.add("John");
              l1.add("Jack");
              System.out.println(l1.size()); //8

              Stream<String> s1=l1.stream();
              Stream<String> s2=s1.distinct();
              long c=s2.count();
              System.out.println(c); //7

              long c1=l1.stream().distinct().count();
              System.out.println(c1); //7,

              boolean b1=l1.stream().distinct().anyMatch((e)->e.startsWith("R"));
              System.out.println(b1);  //true

              boolean b2=l1.stream().distinct().allMatch((e)->e.startsWith("R"));
              System.out.println(b2);  //false

              boolean b3=l1.stream().distinct().noneMatch((e)->e.startsWith("Z"));
              System.out.println(b3);  //true

              List<Student> l2=new ArrayList<>();
              l2.add(new Student(23,"PK"));
              l2.add(new Student(26,"KK"));
              l2.add(new Student(23,"MK"));
              l2.add(new Student(21,"SK"));
              l2.add(new Student(40,"RK"));
              l2.add(new Student(30,"BK"));
              l2.add(new Student(29,"DK"));
              l2.add(new Student(28,"GK"));
              l2.add(new Student(33,"TK"));

              Stream<Student> st=l2.stream().filter((a1)->a1.getId()>25);
              st.forEach(System.out::println);

              Optional opt=Stream.of(3,5,6).reduce((a,b)->a*b);
              System.out.println(opt.get());  //90

              Integer i=Stream.of(3,5,6).reduce(2, (a,b)->a*b);
              System.out.println(i);  //180

              Optional<String> opt1=Stream.of("lion","ape","tiger").min((c11,c21)->c11.length()-c21.length());
              System.out.println(opt1.get());  //ape
            }
}
```

map() - intermediate
- takes one Function functional interface as an argument and returns a stream consisting of result generated by applying the passed function to each element
- used for data transformation

collect() - terminal - used to collect the data as list or set

Collectors.toList()
Collectors.toSet()
Collectors.toMap(Function f1,Function f2)
Collectors.joining()
Collectors.counting()
Collectors.toCollection(Supplier s)
Collectors.partitioningBy() - used to split the list into 2 parts based on true and false
- partitioningBy(predicate)
- partitioningBy(predicate,Collector)

Collectors.groupingBy() - group the stream of elements under some condition
- groupingBy(Function)
- groupingBy(Function,Collector)
- groupingBy(Function,Comparator,Collector)

flatMap() - intermediate
- takes one Function functional interface as an argument and returns a new stream and that stream is copied to another stream which will return the value
- used for data transformation + flatenning

```
public class Main {
            public static void main(String[] args) {
              Integer a[]=new Integer[] {1,2,3,4,5};
              List<Integer> l1=Arrays.asList(a);

              //JDK10
              //List<Integer> l2=List.of(1,2,3,4,5); //create immutable list

              List<Integer> l2=l1.stream().map((e)->e*3).collect(Collectors.toList());
              l2.forEach(System.out::println); //3,6,9,12,15

              List<Integer> l3=l1.stream().flatMap((e1)->Stream.of(e1*2)).collect(Collectors.toList());
               l3.forEach(System.out::println);//2,4,6,8,10

              String s1=Stream.of("one","two","three").collect(Collectors.joining("-"));
              System.out.println(s1);  //one-two-three

              long c=Stream.of("one","two","three").collect(Collectors.counting());
              System.out.println(c);  //3

             List<String> l4=Stream.of("lions","tigers","bears","toads","toads")
                 .filter((s)->s.startsWith("t")).collect(Collectors.toList());
             l4.forEach(System.out::println); //tigers,toads,toads

             Set<String> l5=Stream.of("lions","tigers","bears","toads","toads")
                         .filter((s)->s.startsWith("t")).collect(Collectors.toSet());
                      l5.forEach(System.out::println); //tigers,toads

                      TreeSet<String> ts=Stream.of("lions","tigers","bears","toads","toads","tadpole")
                 .filter((s)->s.startsWith("t")).collect(Collectors.toCollection(TreeSet::new));
                 System.out.println(ts); //[tadpole,tigers,toads]

               Map<String,Integer>  m1=Stream.of("lions","tigers","bears","toads")
                  .collect(Collectors.toMap(k1->k1, String::length));
               m1.forEach((k,v)->System.out.println("Key = "+k+" Value = "+v));

               Map<Boolean,List<String>> m2=Stream.of("lions","tigers","bears","toads","toads","tadpole")
                    .collect(Collectors.partitioningBy((a1)->a1.length()<=5));
               System.out.println(m2);

               Map<Boolean,Set<String>> m3=Stream.of("lions","tigers","bears","toads","toads","tadpole")
                           .collect(Collectors.partitioningBy((a1)->a1.length()<=5,Collectors.toSet()));
                        System.out.println(m3);

                    Map<Integer,List<String>> m4=Stream.of("lions","tigers","bears","lions","ape")
                           .collect(Collectors.groupingBy((e)->e.length()));
                   System.out.println(m4);

                    Map<Integer,Set<String>> m5=Stream.of("lions","tigers","bears","lions","ape")
                           .collect(Collectors.groupingBy((e)->e.length(),Collectors.toSet()));
                   System.out.println(m5);

                    TreeMap<Integer,Set<String>> m6=Stream.of("lions","tigers","bears","lions","ape")
                           .collect(Collectors.groupingBy((e)->e.length(),TreeMap::new,Collectors.toSet()));
                   System.out.println(m6);

            }
}
```

sorted() - intermediate - used to sort the elt

```
public class Main {
            public static void main(String[] args) {
              List<String> l1=List.of("9","A","z","1","B","4","e","f");

              List<String> l21=l1.stream().sorted().collect(Collectors.toList()); //asc order
              System.out.println(l21);
              List<String> l3=l1.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList()); //desc order
              System.out.println(l3);

              List<Student> l2=new ArrayList<>();
               l2.add(new Student(23,"PK"));
               l2.add(new Student(26,"KK"));
               l2.add(new Student(23,"MK"));
               l2.add(new Student(21,"SK"));
               l2.add(new Student(40,"RK"));
               l2.add(new Student(30,"BK"));
               l2.add(new Student(29,"DK"));
               l2.add(new Student(28,"GK"));
               l2.add(new Student(33,"TK"));

               //comparaingInt(), comparaingDouble(), comparaingLong()
```

```
                List<Student> l4=l2.stream().sorted(Comparator.comparingInt(Student::getId))
                                                  .collect(Collectors.toList());
                l4.forEach(System.out::println);
                System.out.println();
                List<Student> l5=l2.stream().sorted(Comparator.comparingInt(Student::getId).reversed())
                        .collect(Collectors.toList());
        l5.forEach(System.out::println);
        System.out.println();
        List<Student> l6=l2.stream().sorted(Comparator.comparing(Student::getName))
                        .collect(Collectors.toList());
         l6.forEach(System.out::println);
         System.out.println();
         List<Student> l7=l2.stream().sorted(Comparator.comparing(Student::getName).reversed())
                        .collect(Collectors.toList());
          l7.forEach(System.out::println);
            }
}
```

Other ways to create stream
1. builder() - create finite stream
2. generate() - create infinite stream
3. iterate() - create infinite stream

Create streams based on primitive datatype
1. IntStream - create stream of int values
2. DoubleStream - create stream of double values
3. LongStream - create stream of long values

From JDK1.9
takeWhile() - if stream does not match the predicate, it will discard the rest of stream
dropWhile() - if stream does not match the predicate, it will print the rest of stream

```
public class Main {
            public static void main(String[] args) {
              Stream<String> s1=Stream.<String>builder().add("Ram").add("Sam").add("Raj").build();
              s1.forEach(System.out::println);

              Stream<String> s2=Stream.generate(()->"hello").limit(5);
              s2.forEach(System.out::println);

              Stream<Integer> s3=Stream.iterate(2, (i)->i*2).skip(3).limit(5);
              s3.forEach(System.out::println);

              IntStream i1=IntStream.range(1,6);  //start to end-1
              i1.forEach(System.out::println);  //1 2 3 4 5

              IntStream i2=IntStream.rangeClosed(1,6);  //start to end
              i2.forEach(System.out::println);  //1 2 3 4 5 6

              IntStream i3="abcd".chars();
              i3.forEach(System.out::println);  //97 98 99 100

              Random r=new Random();
              DoubleStream d1=r.doubles(5);
              d1.forEach(System.out::println);

              List<Student> l2=new ArrayList<>();
               l2.add(new Student(23,"PK"));
               l2.add(new Student(26,"KK"));
               l2.add(new Student(23,"MK"));
               l2.add(new Student(21,"SK"));
               l2.add(new Student(40,"RK"));
               l2.add(new Student(30,"BK"));
               l2.add(new Student(29,"DK"));
               l2.add(new Student(28,"GK"));
               l2.add(new Student(33,"TK"));

               IntStream i5=l2.stream().mapToInt(Student::getId);
               i5.forEach(System.out::println);

              OptionalInt op=l2.stream().mapToInt(Student::getId).max();
              System.out.println(op.getAsInt());  //40

              OptionalDouble op1=l2.stream().mapToDouble(Student::getId).average();
              System.out.println(op1.getAsDouble());

              //IntSummaryStatistics,DoubleSummaryStatistics, LongSummaryStatistics
              IntSummaryStatistics in=l2.stream().collect(Collectors.summarizingInt(t->t.getId()));
              System.out.println(in);
              System.out.println(in.getAverage()+" "+in.getCount());

              Stream.of(2,4,6,8,9,10,12).takeWhile(n->n%2==0).forEach(System.out::println);  //2 4 6 8
            Stream.of(2,4,6,8,9,10,12).dropWhile(n->n%2==0).forEach(System.out::println); //9 10 12

        }
}
```

Java 8 Date API Feature: **[Please refer the notes once I will cover it on Monday once]**

Java 8 introduced the java.time package, which provides a new Date and Time API to address the shortcomings of the previous java.util.Date and java.util.Calendar classes. Here are some advantages of the Java 8 Date API over the previous versions:

Immutability: Objects in the new Date and Time API are immutable, which means once created, their values cannot be changed. This ensures thread safety and reduces potential bugs related to mutable state.

Clarity and Simplicity: The new API provides clear and consistent naming conventions, making it easier to understand and use. It offers classes like LocalDate, LocalTime, LocalDateTime, ZonedDateTime, etc., which are more intuitive than the older classes.

Null Safety: The new API does not allow null values, which helps avoid NullPointerExceptions that were common when working with java.util.Date.

Better API Design: The Java 8 Date API is designed using modern programming principles, such as method chaining, fluent interfaces, and use of enums, which improve code readability and maintainability.

Support for Timezones: The new API provides better support for working with timezones, including classes like ZoneId and ZoneOffset, making it easier to handle timezone-related operations.

Functional Programming Support: The new API integrates well with Java's functional programming features, such as lambdas and streams, allowing for more concise and expressive code when manipulating dates and times.

Backward Compatibility: While the new API is designed to address the limitations of the old Date and Calendar classes, it also provides methods to convert between the old and new date/time representations, ensuring backward compatibility with existing code.

Drawbacks of the previous versions (Java 7 and earlier) include:

Mutability: java.util.Date and java.util.Calendar classes are mutable, leading to potential thread-safety issues in multi-threaded environments.

Inconsistent API: The old API had inconsistent and confusing method names, leading to difficulties in understanding and using the classes effectively.

Lack of Null Safety: java.util.Date allows null values, which can lead to NullPointerExceptions if not handled properly.

Limited Timezone Support: Working with timezones was cumbersome and error-prone in the old API, lacking adequate support for timezone-related operations.

Verbosity: Code using the old API often became verbose and hard to read due to the complex nature of date and time manipulation.

Overall, the Java 8 Date and Time API provides significant improvements over the previous versions, addressing many of the limitations and shortcomings of the old java.util.Date and java.util.Calendar classes.

```
import java.time.*;
import java.time.format.*;
import java.util.*;
import java.util.stream.*;

public class DateAPIExample {
    public static void main(String[] args) {
        // Advantages of Java 8 Date API
        // 1. Immutability
        LocalDate date = LocalDate.now();
        LocalDate modifiedDate = date.plusDays(1);

        // 2. Clarity and Simplicity
        LocalDateTime dateTime = LocalDateTime.of(2024, Month.FEBRUARY, 2, 10, 30);

        // 3. Null Safety
        try {
            LocalDate nullDate = null; // This would cause a NullPointerException
            System.out.println(nullDate);
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught due to null date.");
        }

        // 4. Better API Design
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime future = now.plusDays(3).minusHours(2);

        // 5. Support for Timezones
        ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of("America/New_York"));

        // 6. Functional Programming Support
        List<LocalDate> datesInRange =
            Stream.iterate(LocalDate.now(), dateIter -> dateIter.plusDays(1))
                  .limit(10)
                  .collect(Collectors.toList());
        // 7. Backward Compatibility
        Date oldDate = new Date();
        LocalDate newDate = oldDate.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();

        // Drawbacks of Previous Versions
        // 1. Mutability
        Date mutableDate = new Date();
        mutableDate.setTime(System.currentTimeMillis() + 10000); // Changing the time

        // 2. Inconsistent API
        Calendar calendar = Calendar.getInstance();
        int year = calendar.get(Calendar.YEAR);

        // 3. Lack of Null Safety
        Date nullableDate = null; // This is allowed

        // 4. Limited Timezone Support
        Calendar calendarNY = Calendar.getInstance(TimeZone.getTimeZone("America/New_York"));

        // 5. Verbosity
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        Date parsedDate;
        try {
            parsedDate = sdf.parse("2024-02-02");
            System.out.println(parsedDate);
        } catch (ParseException e) {
            System.out.println("Parsing exception: " + e.getMessage());
        }
    }
}
```

Regards,
Oviya Prakasam

Subject: RE: USAA - Java Features Training - Day 2 Notes

Hi Team,

Please find Day2 Notes:

Hands-On:

1. **Java static method & Default method implementation. Use Book Object for reference.**
2. **Implement method reference using Custom Math class (Arithmetic operations) and perform operations using method reference. (Add, Sub, Division) – Method static ref, Non Static (Instance reference), Constructor reference**
3. **Create an Employee Object and store the object values in a list with some empty and proper employee values (EmpId, EmpName, EmpRole) then check the value is null or valid and print else use custom exception if not found (Optional methods, IsPresent, Nullable, OrElse, OrElseThrow)**

Method Reference in Java - 8:

Java provides a new feature called *method reference* in Java 8. **Method reference is used to refer method of the functional interface.** It is a compact and easy form of a lambda expression. Each time when you are using a **lambda expression** to just referring a method, you can **replace your lambda expression with a method reference**.
In this post, we will learn what is method reference, what are their benefits, and what are different types of method references.

Kinds of Method References

There are four kinds of method references:

Example 1: This example shows, how the lambda expression replaced with method refers to the **static** method.

```
package com.java.lambda;

import java.util.*;
import java.util.function.BiFunction;
import java.util.function.Function;

public class MethodRefDemo {

    public static int addition(int a, int b){
        return (a + b);
    }

    public static void main(String[] args) {
        // 2. Method reference to a static method of a class
        Function<Integer, Double> sqrt = (Integer input) -> Math.sqrt(input);
        System.out.println(sqrt.apply(4));
        Function<Integer, Double> sqrtRef = Math::sqrt;
        System.out.println(sqrtRef.apply(4));

        BiFunction<Integer, Integer, Integer> functionLambda = (a, b) -> MethodRefDemo.addition(a, b);
        System.out.println(functionLambda.apply(10, 20));

        BiFunction<Integer, Integer, Integer> function = MethodRefDemo::addition;
        System.out.println(function.apply(10, 20));
    }
}
```

2. Reference to an Instance Method

Example 1: In the following example, we are referring to non-static methods. You can refer to methods by a class object and anonymous object.

```
public class ReferenceToInstanceMethod {

    public void saySomething() {
        System.out.println("Hello, this is non-static method.");
    }

    public static void main(String[] args) {
        // Creating object
        ReferenceToInstanceMethod methodReference = new ReferenceToInstanceMethod();
        // Referring non-static method using reference
        Sayable sayable = methodReference::saySomething;
        // Calling interface method
        sayable.say();
        // Referring non-static method using anonymous object

        // You can use  anonymous object also
        Sayable sayable2 = new ReferenceToInstanceMethod()::saySomething;
        // Calling interface method
        sayable2.say();
    }
}

interface Sayable {
    void say();
}
```

Example 3: In the following example, we are using *BiFunction* interface. It is a predefined interface and contains a functional method to *apply()*. Here, we are referring to add a method to apply method.

```
import java.util.function.BiFunction;

class Arithmetic {
    public int add(int a, int b) {
        return a + b;
    }
}

public class InstanceMethodReference3 {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> adder = new Arithmetic()::add;
       int result = adder.apply(10, 20);
       System.out.println(result);
   }
}
```

3. Reference to an Instance Method of an Arbitrary Object of a Particular Type

Sometimes, we call a method of argument in the lambda expression. In that case, we can use a method reference to call an instance method of an arbitrary object of a specific type.

Syntax :

```
ContainingType::methodName
```

Example: The following is an example of a reference to an instance method of an arbitrary object of a particular type:

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
    "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

4. Reference to a Constructor

You can refer to a constructor by using the new keyword. Here, we are referring constructor with the help of a functional interface.
Syntax :

```
ClassName::new
```

Example 1: The below example demonstrates the usage of method reference to the constructor.

```
public class ReferenceToConstructor {
   public static void main(String[] args) {
       Messageable hello = Message::new;
       hello.getMessage("Hello");
   }
}

interface Messageable{
   Message getMessage(String msg);
}

class Message{
   Message(String msg){
      System.out.print(msg);
   }
}
```

Example 2: Write the Lambda to convert List into Set and convert Lambda into method reference:

```
      // 4. reference to a constructor
      List<String> fruits = new ArrayList<>();
      // Adding new elements to the ArrayList
      fruits.add("Banana");
      fruits.add("Apple");
      fruits.add("mango");
      fruits.add("orange");

      // Using lambda expression
      Function<List<String>, Set<String>> f2 = (nameList) -> new HashSet<>(nameList);
      Set<String> set2 = f2.apply(fruits);
      System.out.println(set2);

      // Using Method reference
      Function<List<String>,Set<String>> f3= HashSet::new;
      Set<String> set = f3.apply(fruits);
      System.out.println(set);
```

2. Java 8 Optional Classes:

Java introduced a new class Optional in jdk8.

It is a public final class and used to deal with NullPointerException in Java application. You must

import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

Example:

Before Java 8:

Example: Java Program without using Optional

In the following example, we are not using Optional class. This program terminates abnormally and throws a nullPointerException.

```
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      String lowercaseString = str[5].toLowerCase();
      System.out.print(lowercaseString);
   }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException
        at lambdaExample.OptionalExample.main(OptionalExample.java:6)
```

After Java 8:

```
class Book {
   private String title;
   private String author;
   private double price;

   public Book(String title, String author, double price) {
      this.title = title;
      this.author = author;
      this.price = price;
   }

   public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public double getPrice() {
        return price;
    }
}

class Library {
    public Optional<Book> findBook(String title) {
        // In a real scenario, this method would search the library database
        // and return an Optional containing the Book if found, or an empty Optional if not found
        if (title.equals("Java Programming")) {
            return Optional.of(new Book("Java Programming", "John Doe", 29.99));
        } else {
            return Optional.empty();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Library library = new Library();

        // Example 1: Retrieve book information if present
        Optional<Book> optionalBook = library.findBook("Java Programming");
        optionalBook.ifPresent(book -> System.out.println("Book found: " + book.getTitle() + " by " + book.getAuthor()));

        // Example 2: Retrieve book price, or default to 0 if book not found
        double price = library.findBook("Java Programming").map(Book::getPrice).orElse(0.0);
        System.out.println("Price of the book: $" + price);

        // Example 3: Retrieving book author, or throw exception if book not found
        String author = library.findBook("Java Programming").map(Book::getAuthor)
                               .orElseThrow(() -> new RuntimeException("Book not found"));
        System.out.println("Author of the book: " + author);

        // Example 4: Chaining multiple Optional operations
        double discountedPrice = library.findBook("Java Programming")
                                .map(Book::getPrice)
                                .filter(price1 -> price1 > 20) // Apply discount only if price > $20
                                .map(price1 -> price1 * 0.9) // Apply 10% discount
                                .orElse(0.0); // If book not found or price <= $20, default to 0
        System.out.println("Discounted price of the book: $" + discountedPrice);
    }
}
```

https://www.javatpoint.com/java-8-optional

---

3. Default Methods in Interface Examples

To better understand the functionality of default interface methods, let’s create a simple example.

Step 1: Creare Vehicle interface and just one implementation. Let's first discuss just one implementation of default methods of Vehicle interface.

```
public interface Vehicle {
String getBrand();

String speedUp();

String slowDown();

default String turnAlarmOn() {
  return "Turning the vehice alarm on.";
}

default String turnAlarmOff() {
  return "Turning the vehicle alarm off.";
}

static int getHorsePower(int rpm, int torque) {
  return (rpm * torque) / 5252;
}
}
```

Step 2: let’s write the implementing class for Vehicle interface.

```
public class Car implements Vehicle {

    private final String brand;

    public Car(String brand) {
        this.brand = brand;
    }

    @Override
    public String getBrand() {
        return brand;
    }

    @Override
    public String speedUp() {
        return "The car is speeding up.";
    }

    @Override
    public String slowDown() {
        return "The car is slowing down.";
    }
}
```

Step 3: Let's test the above implementation with main() method.
Please notice how the default methods turnAlarmOn() and turnAlarmOff() from our Vehicle interface are automatically available in the Car class.

```
public class TestJava8Interface {
public static void main(String[] args) {

  Vehicle car = new Car("BMW");
  System.out.println(car.getBrand());
  System.out.println(car.speedUp());
  System.out.println(car.slowDown());
  System.out.println(car.turnAlarmOn());
  System.out.println(car.turnAlarmOff());
  System.out.println(Vehicle.getHorsePower(2500, 480));

 }
}
```

Output :

```
BMW
The car is speeding up.
The car is slowing down.
Turning the vehice alarm on.
Turning the vehicle alarm off.
```

Now, we write two implementations for Vehicle interface and test the default methods behavior.
Let's create Motorbike class which implements Vehicle interface.

```
public class Motorbike implements Vehicle {

    private final String brand;

    public Motorbike(String brand) {
        this.brand = brand;
    }

    @Override
    public String getBrand() {
        return brand;
    }

    @Override
    public String speedUp() {
        return "The motorbike is speeding up.";
    }

    @Override
    public String slowDown() {
        return "The motorbike is slowing down.";
    }
}
```

Let's test the two implementations for Vehicle interface with main() method.
The most typical use of default methods in interfaces is to incrementally provide additional functionality to a given type without breaking down the implementing classes.

```
public class TestJava8Interface {
public static void main(String[] args) {

  Vehicle car = new Car("BMW");
  System.out.println(car.getBrand());
  System.out.println(car.speedUp());
  System.out.println(car.slowDown());
  System.out.println(car.turnAlarmOn());
 System.out.println(car.turnAlarmOff());
  System.out.println(Vehicle.getHorsePower(2500, 480));

  Vehicle bike = new Motorbike("ACTIVA 4G");
  System.out.println(bike.getBrand());
  System.out.println(bike.speedUp());
  System.out.println(bike.slowDown());
  System.out.println(bike.turnAlarmOn());
  System.out.println(bike.turnAlarmOff());
  System.out.println(Vehicle.getHorsePower(2500, 480));

 }
}
```

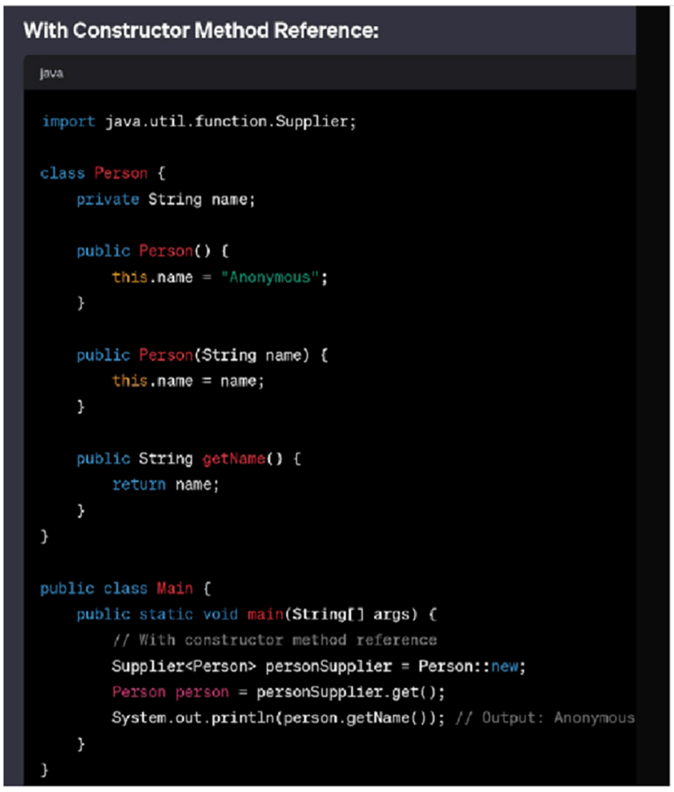
Generally, static methods are used to define utility methods.
The idea behind static interface methods is to provide a simple mechanism that allows us to increase the degree of cohesion of a design by putting together related methods in one single place without having to create an object.
Furthermore, static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.
4.1 Example of a static method in Interface

```
public interface Vehicle {
String getBrand();

String speedUp();

String slowDown();

default String turnAlarmOn() {
  return "Turning the vehice alarm on.";
}

default String turnAlarmOff() {
  return "Turning the vehicle alarm off.";
}

static int getHorsePower(int rpm, int torque) {
  return (rpm * torque) / 5252;
}
}
```

There is getHorsePower(int, int) static method in Vehicle interface.
Let's see how the client uses this method :

```
public class TestJava8Interface {
public static void main(String[] args) {

  Vehicle car = new Car("BMW");
  System.out.println(car.getBrand());
  System.out.println(car.speedUp());
  System.out.println(car.slowDown());
  System.out.println(car.turnAlarmOn());
  System.out.println(car.turnAlarmOff());
  System.out.println(Vehicle.getHorsePower(2500, 480));
}
}
```

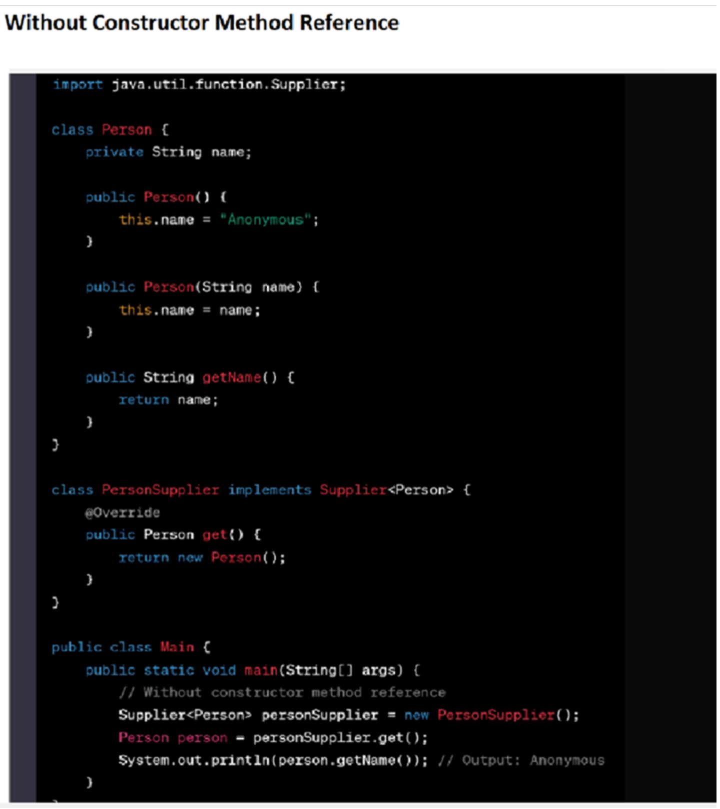
Additional Notes for constructor method reference for the doubts raised:



With Constructor Method Reference:

java

```java
import java.util.function.Supplier;

class Person {
    private String name;

    public Person() {
        this.name = "Anonymous";
    }

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        // With constructor method reference
        Supplier<Person> personSupplier = Person::new;
        Person person = personSupplier.get();
        System.out.println(person.getName()); // Output: Anonymous
    }
}
```



Without Constructor Method Reference

```java
import java.util.function.Supplier;

class Person {
    private String name;

    public Person() {
        this.name = "Anonymous";
    }

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

class PersonSupplier implements Supplier<Person> {
    @Override
    public Person get() {
        return new Person();
    }
}

public class Main {
    public static void main(String[] args) {
        // Without constructor method reference
        Supplier<Person> personSupplier = new PersonSupplier();
        Person person = personSupplier.get();
        System.out.println(person.getName()); // Output: Anonymous
    }
```

Regards,
Oviya Prakasam

Subject: RE: USAA - Java Features Training - Day 1 Notes

Hi Team,

Please find Day1 Notes:

Hands-On:

1. Implement 1 functional Interface and implement the method with using lambda expression and without lambda expression
2. Implement Employee Object with Map() and Filter() stream API methods.
3. Implement any 2 Predefined functional interfaces and understand how to implement with and without lambda expression.

Java 8 Features and Java 17 Features List
- Implementations on few Java 8 Concepts.
1. Stream API
2. Functional Interface - only one abstract methods
3. Lambda expr - used to enable functional prg in java - we can write functions that exists on its own
- Java 8 Features additional topic coverage.
4. Method reference using ::
5. Date API
6. Optional class
7. Default and static method inside interface

- Java 17 features Implementation coverage

Pattern Matching For Switch Statements
Traditional Switch Statement
Switch Statements Since Java 12
Switch Statements Since Java 17
Pattern Matching for instanceof
Sealed Classes & Interfaces

1. Functional Interfaces: Before
[Functional Interfaces in Java - GeeksforGeeks]
java Copy code
```
  // Before Java 8
  public interface Calculator {
 int calculate(int a, int b);
}

public class CalculatorImpl implements Calculator {
 public int calculate(int a, int b) {
  return a + b;
 }
}
```
After :

```
  java Copy code
  // Java 8
  @FunctionalInterface public interface Calculator {
 int calculate(int a, int b);
}

public class CalculatorMain {
 public static void main(String[] args) {
  Calculator addition = (a, b) -> a + b;
  System.out.println(addition.calculate(3, 5));
 }
}
```

2. Lambda Expressions: Before:
```
  java Copy code
  // Before Java 8
  public class HelloWorld {
 public static void main(String[] args) {
  Greeting greeting = new Greeting() {
   public void greet() {
    System.out.println("Hello, World!");
   }
  };
  greeting.greet();
 }
}

interface Greeting {
 void greet();
}
```
After :

```
  java Copy code
  // Java 8
  public class HelloWorld {
 public static void main(String[] args) {
  Greeting greeting = () -> System.out.println("Hello, World!");
  greeting.greet();
  }
}

interface Greeting {
  void greet();
}
```

Lambda expression : are just a funcƟon that exists on its own and execute it
independently,
used to enable funcƟonal programming

How to write Lambda expr?

1. We take a funcƟon and assign to a variable a = public void perform() {
 System.out.println("Hello");
}
- public,
 private, protected make sense if we write any method inside class,
 but in lambda expr,
 funcƟons are exists on its own so when we write lambda expr there is no need
 to define access specifiers
2. a = void perform() {
 System.out.println("Hello");
}
Whenever we assign a funcƟon to a variable,
 we can access the funcƟon using ur variable name,
 so when we write lambda expr there is no need to define funcƟon name 3. a =
 void() {
 // System.out.println("Hello");
 return "hello";
}
By seeing the funcƟon itself we can idenƟfy the return type of the funcƟon,
 so when we write lambda expr there is no need to define
 its return type 4. a = () {
 System.out.println("Hello");
}
Lambda expr contains parenthesis for input arg, logic and we need to put ->symbol
a=() -> {
 System.out.println("Hello");
}
b = (int a, int b) -> {
 System.out.println(a + b);
}
5. FuncƟonType<Void, Void> a = () -> {
 System.out.println("Hello");
}
FuncƟonalInterface a = () -> {
 System.out.println("Hello");
}
Use a funcƟonal interface as a return type for lambda expr, this concept is called Type Inference
- Lambda expr is always used to logic only for methods in funcƟonal interface
Rules
1. If ur body of lambda expr is just a single line,
 then we can ignore curly braces FuncƟonalInterface a =
 () -> System.out.println("hello");
2. If ur body of lambda expr is just a single line,
 then we can ignore the return stmt FuncƟonalInterface b = (int a) -> {
 return a * 2;
}
FuncƟonalInterface b = (int a) -> a * 2;

Lambda expression Example:

```
package com.hcl.test;

@FunctionalInterface
interface Calculator {
        //int calculate(int a, int b);
        String mynameMethod(String printName);
}

public class Java8Features implements Calculator{

        public static void main(String[] args) {

                Java8Features withoutLamda = new Java8Features();
                System.out.println(withoutLamda.mynameMethod(" My Name "));

                //using Lambda
                Calculator printname =  (name) -> name;
                System.out.println(printname.mynameMethod(" My Name "));
        }

        //Without using lambda
        @Override
        public String mynameMethod(String printName) {
                return printName;
        }
}
```

3. Streams API:

Description:

Streams API
 - contains collection of objects from some sources(Array,List,Set) and process them sequentially
- Collection framework also stores collection of objects and we can manipulate the object, but streams are only used for processing the data and we cant manipulate the data

- present in java.util.stream.* package
- 2 types of streams - finite(fixed number of values) and infinite(unlimited) stream
 - 2 types of operation

1. Intermediate operation - return stream itself - optional operation - we can chain multiple intermediate operation
- filter(), map(), flatMap(), sorted(), peek(), distinct(), limit(), skip()

2. Terminal operation - It will traverse into the newly generated stream and return single value - mandatory operation - we cant chain terminal operation

- toArray(), forEach(), count(), min(), max(), reduce(), collect(),anyMatch(), allMatch(), noneMatch(), findFirst()

3 steps

Creation of Stream - Intermediate operation - Terminal Operation
- If we perform terminal operation then that stream is completely closed, so when we perform any other operation on closed stream then we will get IllegalStateException

1. Creation of Stream - Finite stream - 2 ways

1. stream() - used to generate a stream from some source (ie) array or list or set

```
String s[]=new String[]{"one","two","three"};
Stream<String> s1=s.stream();
List<String> l1=new ArrayList<>();
l1.add("Ram");
l1.add("Sam");
l1.add("Raj");
Stream<String> s2=l1.stream();
```

2. of() - used to create our own stream

```
Stream<Integer> st= Stream.of(1,2,3,4,5);
```

filter() - intermediate - used to filter the data based on some condition

```
package com.hcl.test;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamsAPI {
  public static void main(String[] args) {
    // Creating a list of employees

    List<Employee1> employees = Arrays.asList(new Employee1("Alice", 30, 50000),
        new Employee1("Bob", 25, 60000), new Employee1("Charlie", 35, 70000),
        new Employee1("David", 40, 80000), new Employee1("Eva", 28, 55000));

    // Example 1: forEach
    System.out.println("Example 1: forEach");
    employees.stream().forEach(System.out::println);

    // Class Name :: methodname

    for (Employee1 emp : employees) {
      System.out.println(emp);
    }

    System.out.println("\nExample 3: filter");
    List<Employee1> filtered = employees.stream()
                              .filter(e -> e.getEmpSal() > 60000)
                              .collect(Collectors.toList());
    System.out.println("Employees with salary > 60000: " + filtered);

    System.out.println("\nExample 4: limit");
    List<Employee1> limited =
        employees.stream().limit(3).collect(Collectors.toList());
    System.out.println("First 3 employees: " + limited);

    // List<String> name = employees.stream().map(Employee1::getName).

    List<String> names = employees.stream()
                         .map(Employee1::getEmpName)
                         .collect(Collectors.toList());

    //.map(Employee1::getEmpName).collect(Collectors.toList());
    System.out.println("Employee names: " + names);
  }
}
```

Regards,
Oviya Prakasam

-----Original Appointment-----
From: Oviya P <oviya.p@hcl.com>
Sent: Wednesday, January 31, 2024 7:44 AM
To: Shaik Aavez; Madgula Gayathri; Avula Manohar; Akash T; Amlipur Sribhavani; Jakkani Pravalika; Gunasani Vinay; Bemisha J V; Krithiga G
Cc: Anil Abraham, Chennai; Balamuralikrishnan Vengateson
Subject: USAA - Java Features Training
When: Occurs every Monday, Tuesday, Wednesday, Thursday, and Friday effective 1/31/2024 until 2/2/2024 from 10:00 AM to 11:00 AM (UTC+05:30) Chennai, Kolkata, Mumbai, New Delhi.
Where: Microsoft Teams Meeting

Hi Team,

Please find the below schedule for Java features Training.

Java Features

Day/Date	**Topic to be discussed**
Jan 31st	Java 8 Features and Java 17 Features List - Implementations on few Java 8 Concepts. 1. Stream API 2. Functional Interface - only one abstract methods 3. Lambda expr - used to enable functional prg in java - we can write functions that exists on its own - Java 8 Features additional topic coverage. 4. Method reference using :: 5. Date API 6. Optional class 7. Default and static method inside interface
Feb 1 & 2nd st	- Java 17 features Implementation coverage Pattern Matching For Switch Statements Traditional Switch Statement Switch Statements Since Java 12 Switch Statements Since Java 17 Pattern Matching for instanceof Sealed Classes & Interfaces
Feb 5th	- Java Other version features List with theoratical coverage.

Regards,
Oviya Prakasam