

AIM:

To demonstrate intermediate stream operations such as `map()`, `filter()`, `sorted()`, and `distinct()` using Java Streams API.

ALGORITHM:

1. Create a list of integers.
2. Convert the list into a stream.
3. Use `filter()` to select even numbers.
4. Use `map()` to square the filtered numbers.
5. Use `distinct()` to remove duplicates.
6. Use `sorted()` to sort the result.
7. Display the output.

PROCEDURE:

1. Import required packages.
2. Create a list using `Arrays.asList()`.
3. Apply stream intermediate operations.
4. Print the processed stream using `forEach()`.

PROGRAM:

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

public class StreamIntermediateDemo {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(2, 5, 3, 6, 2, 8, 5);

        numbers.stream()
            .filter(n -> n % 2 == 0)
```

```

        .map(n -> n * n)

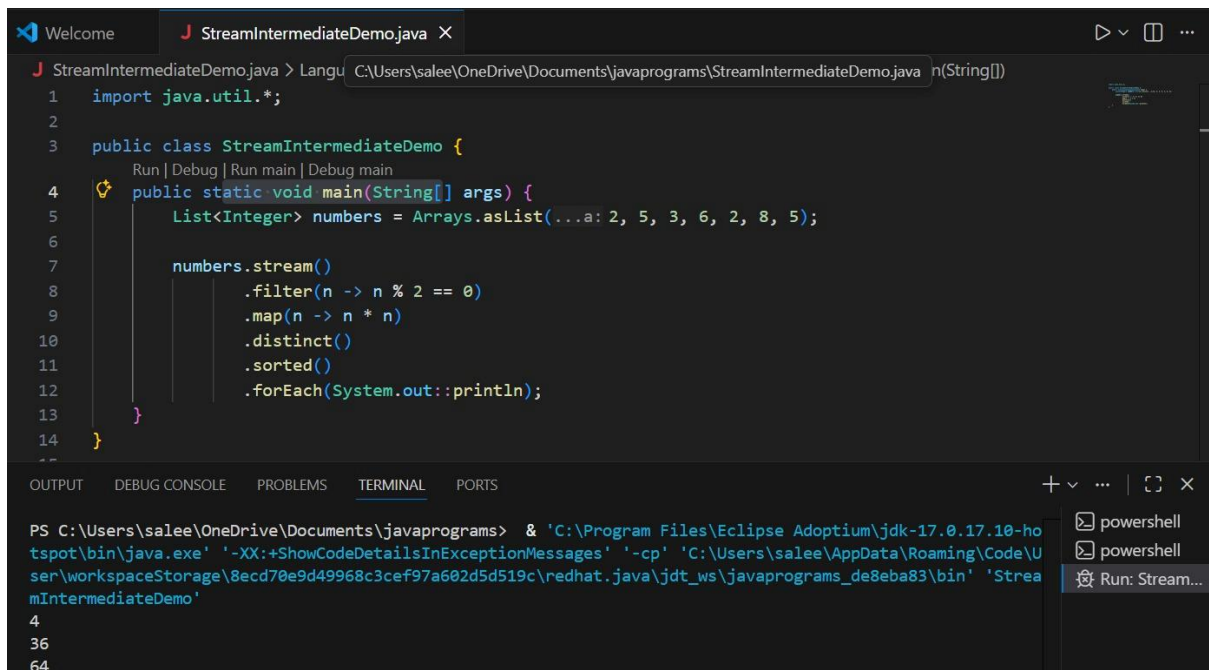
        .distinct()

        .sorted()

        .forEach(System.out::println);
    }
}

```

OUTPUT:



The screenshot shows an IDE window with a Java file named `StreamIntermediateDemo.java`. The code defines a class `StreamIntermediateDemo` with a `main` method. Inside `main`, a list of integers is created: `List<Integer> numbers = Arrays.asList(2, 5, 3, 6, 2, 8, 5);`. This list is then processed using Java Streams: `numbers.stream().filter(n -> n % 2 == 0).map(n -> n * n).distinct().sorted().forEach(System.out::println);`. The IDE's terminal shows the command to run the program, and the output displays the results of the stream operations: `4`, `36`, and `64`.

```

J StreamIntermediateDemo.java > Language C:\Users\salee\OneDrive\Documents\javaprograms\StreamIntermediateDemo.java n(String[])
1  import java.util.*;
2
3  public class StreamIntermediateDemo {
4      public static void main(String[] args) {
5          List<Integer> numbers = Arrays.asList(2, 5, 3, 6, 2, 8, 5);
6
7          numbers.stream()
8              .filter(n -> n % 2 == 0)
9              .map(n -> n * n)
10             .distinct()
11             .sorted()
12             .forEach(System.out::println);
13     }
14 }

```

OUTPUT DEBUG CONSOLE PROBLEMS TERMINAL PORTS

```

PS C:\Users\salee\OneDrive\Documents\javaprograms> & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.17-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\salee\AppData\Roaming\Code\User\workspaceStorage\8ecd70e9d49968c3cef97a602d5d519c\redhat.java\jdt_ws\javaprograms_de8eba83\bin' 'StreamIntermediateDemo'
4
36
64

```

RESULT:

The program successfully demonstrates the use of **intermediate Stream operations** such as `filter()`, `map()`, `distinct()`, and `sorted()`. It shows how data can be processed step-by-step in a functional manner using Java Streams to produce the desired output efficiently.

AIM:

To demonstrate terminal operations of Stream API such as collect(), count(), reduce(), and anyMatch().

ALGORITHM:

1. Create a list of integers.
2. Convert the list into a stream.
3. Use count() to count elements.
4. Use reduce() to find sum.
5. Use anyMatch() to check a condition.
6. Use collect() to store filtered results into a list.
7. Display outputs.

PROCEDURE:

1. Import required packages.
2. Create a list of integers.
3. Apply terminal operations.
4. Print the results.

PROGRAM:

```
import java.util.*;

import java.util.stream.*;

public class StreamTerminalDemo {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
```

```
long count = numbers.stream().count();  
System.out.println("Count: " + count);
```

```
int sum = numbers.stream()  
    .reduce(0, Integer::sum);  
System.out.println("Sum: " + sum);
```

```
boolean hasGreaterThan40 =  
    numbers.stream().anyMatch(n -> n > 40);  
System.out.println("Any number > 40: " +  
hasGreaterThan40);
```

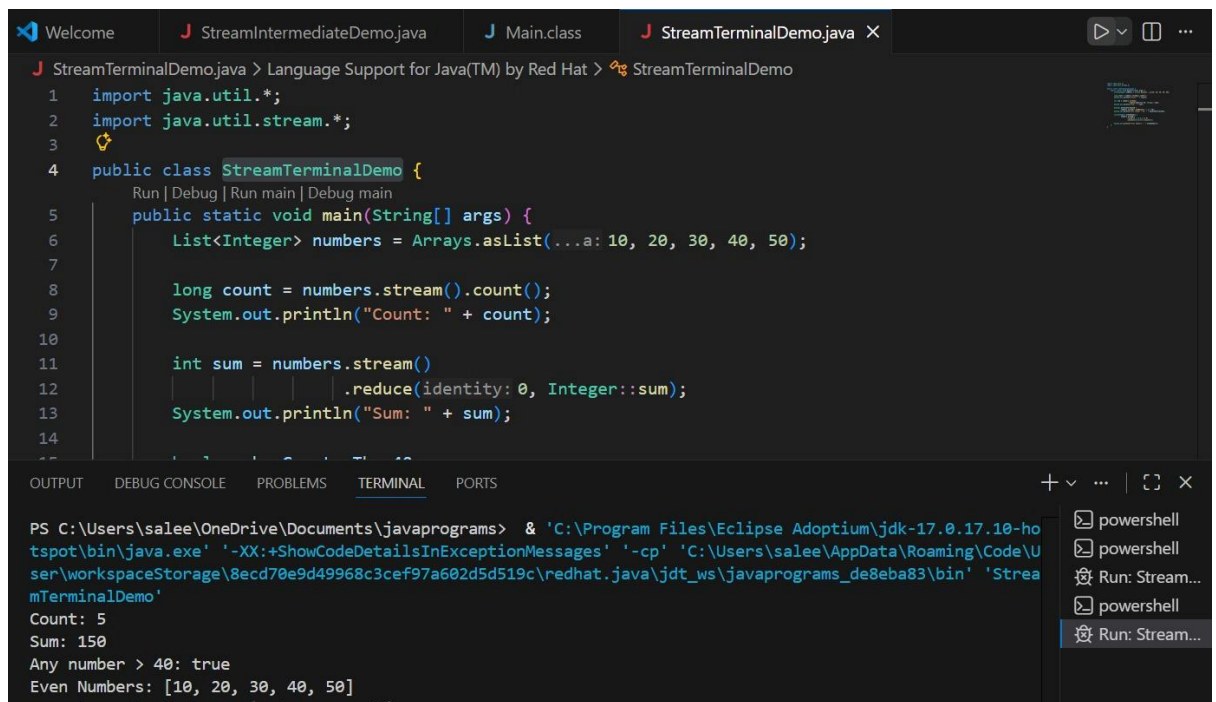
```
List<Integer> evenNumbers =  
    numbers.stream()  
        .filter(n -> n % 2 == 0)  
        .collect(Collectors.toList());
```

```
System.out.println("Even Numbers: " + evenNumbers);
```

```
}
```

```
}
```

OUTPUT:



The screenshot shows the Eclipse IDE with a Java file named `StreamTerminalDemo.java`. The code defines a `StreamTerminalDemo` class with a `main` method. The `main` method creates a list of integers `[10, 20, 30, 40, 50]`, then uses `stream().count()` to count the elements, `stream().reduce(identity: 0, Integer::sum)` to calculate the sum, and `anyMatch()` to check if any number is greater than 40. The output of the program is displayed in the terminal window at the bottom.

```
1 import java.util.*;
2 import java.util.stream.*;
3
4 public class StreamTerminalDemo {
5     public static void main(String[] args) {
6         List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
7
8         long count = numbers.stream().count();
9         System.out.println("Count: " + count);
10
11         int sum = numbers.stream()
12             .reduce(identity: 0, Integer::sum);
13         System.out.println("Sum: " + sum);
14     }
15 }
```

OUTPUT

```
PS C:\Users\salee\OneDrive\Documents\javaprograms> & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.17-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\salee\AppData\Roaming\Code\User\workspaceStorage\8ecd70e9d49968c3cef97a602d5d519c\redhat.java\jdt_ws\javaprograms_de8eba83\bin' 'StreamTerminalDemo'
Count: 5
Sum: 150
Any number > 40: true
Even Numbers: [10, 20, 30, 40, 50]
```

RESULT:

The program effectively demonstrates terminal Stream operations including `count()`, `reduce()`, `anyMatch()`, and `collect()`. It confirms that Stream terminal operations are used to produce final results from a stream pipeline.

AIM:

To demonstrate the use of `ZonedDateTime` and `ZoneId` classes for handling date and time with time zones.

ALGORITHM:

1. Obtain the current date and time.
2. Create `ZoneId` objects for different regions.
3. Use `ZonedDateTime.now()` with different zones.
4. Display date and time for each zone.

PROCEDURE:

1. Import `java.time` package.
2. Create `ZoneId` for required countries.
3. Fetch date and time using `ZonedDateTime`.
4. Display the output.

PROGRAM:

```
import java.time.*;
```

```
public class ZonedDateTimeDemo {
```

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

```
        ZoneId indiaZone = ZoneId.of("Asia/Kolkata");
```

```
        ZoneId usaZone = ZoneId.of("America/New_York");
```

```
        ZoneId ukZone = ZoneId.of("Europe/London");
```

```
        ZonedDateTime indiaTime = ZonedDateTime.now(indiaZone);
```

```

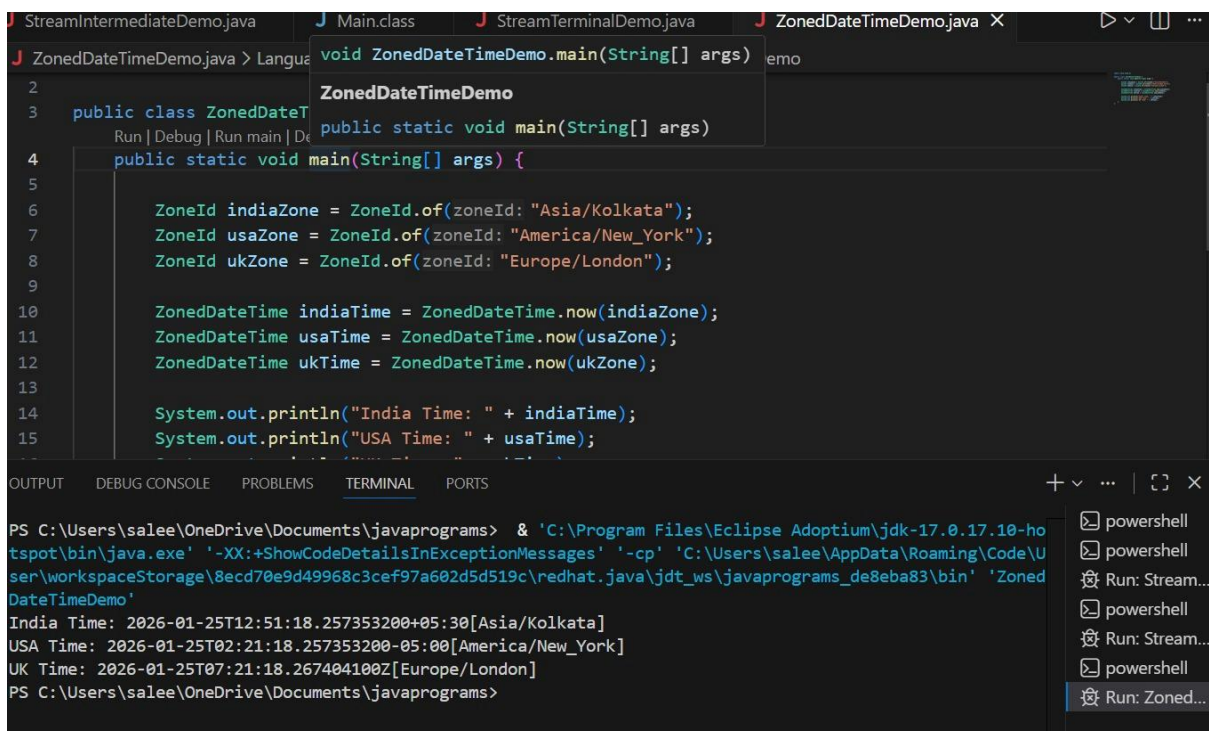
        ZonedDateTime usaTime = ZonedDateTime.now(usaZone);

        ZonedDateTime ukTime = ZonedDateTime.now(ukZone);


        System.out.println("India Time: " + indiaTime);
        System.out.println("USA Time: " + usaTime);
        System.out.println("UK Time: " + ukTime);
    }
}

```

OUTPUT:



The screenshot shows an IDE with a Java file named `ZonedDateTimeDemo.java`. The code defines a `ZonedDateTimeDemo` class with a `main` method that creates `ZoneId` objects for India, USA, and UK, then uses `ZonedDateTime.now()` to get the current time for each zone. The output is printed to the console.

```

void ZonedDateTimeDemo.main(String[] args)
ZonedDateTimeDemo
public static void main(String[] args)
public static void main(String[] args) {
    ZoneId indiaZone = ZoneId.of(zoneId: "Asia/Kolkata");
    ZoneId usaZone = ZoneId.of(zoneId: "America/New_York");
    ZoneId ukZone = ZoneId.of(zoneId: "Europe/London");

    ZonedDateTime indiaTime = ZonedDateTime.now(indiaZone);
    ZonedDateTime usaTime = ZonedDateTime.now(usaZone);
    ZonedDateTime ukTime = ZonedDateTime.now(ukZone);

    System.out.println("India Time: " + indiaTime);
    System.out.println("USA Time: " + usaTime);
}

```

The terminal output shows the execution of the program:

```

PS C:\Users\salee\OneDrive\Documents\javaprograms> & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.17.10-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\salee\AppData\Roaming\Code\User\workspaceStorage\8ecd70e9d49968c3cef97a602d5d519c\redhat.java\jdt_ws\javaprograms_de8eba83\bin' 'ZonedDateTimeDemo'
India Time: 2026-01-25T12:51:18.257353200+05:30[Asia/Kolkata]
USA Time: 2026-01-25T02:21:18.257353200-05:00[America/New_York]
UK Time: 2026-01-25T07:21:18.267404100Z[Europe/London]
PS C:\Users\salee\OneDrive\Documents\javaprograms>

```

RESULT:

The program successfully demonstrates the use of `ZonedDateTime` and `ZoneId` classes to display date and time for different time zones. It highlights how Java's Date and Time API handles global time zone differences accurately.