Lesson 06 Demo 09

Implementing Streams in Java

Objective: To implement streams in Java to perform functional-style operations on collections and improve code readability and efficiency.

Tools Required: Eclipse IDE

Prerequisites: None

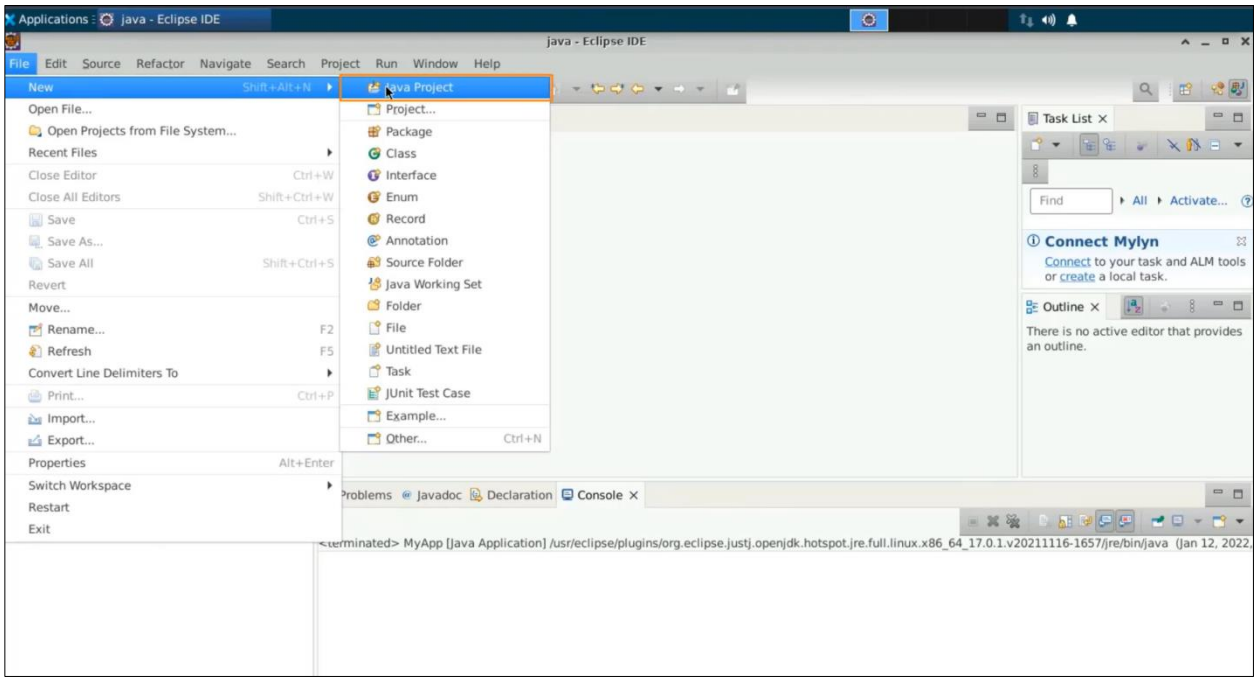
Steps to be followed:

1. Create a new Java project
2. Create one of the stream objects
3. Create a stream using various approaches
4. Iterate the streams and execute the code
5. Create a stream of integer numbers and execute the code
6. Generate another stream and re-execute the code

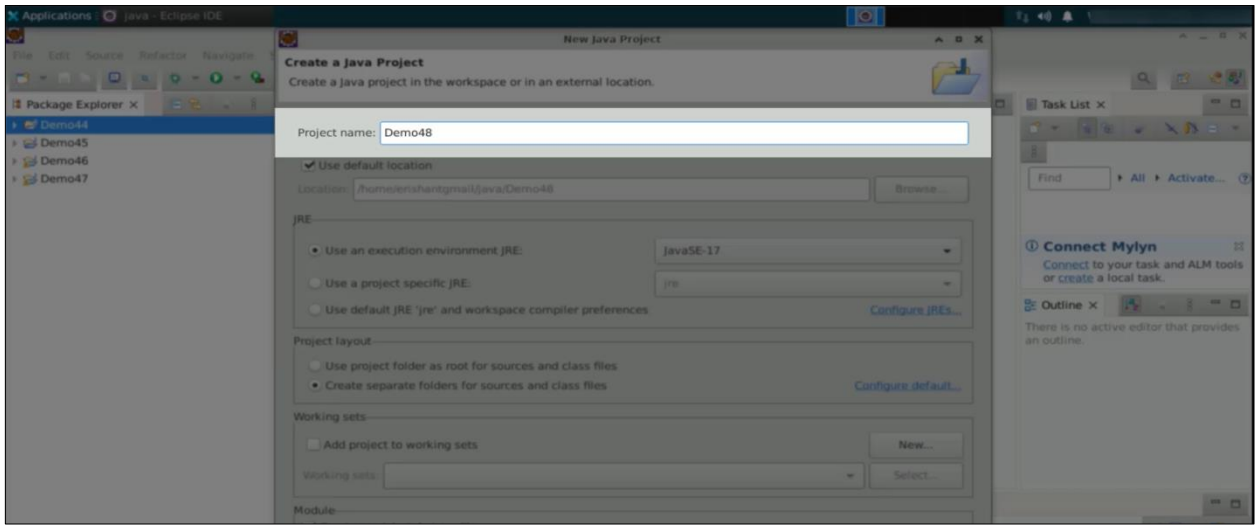
Step 1: Create a new Java project

1.1 Open the **Eclipse IDE**

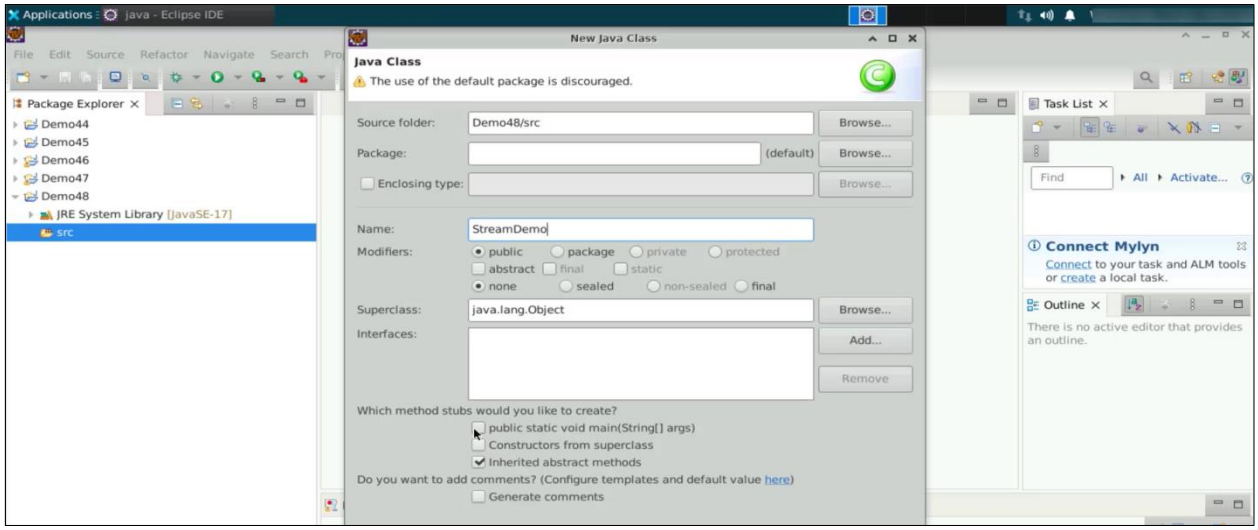
1.2 Select **File**, then **New,** and click **Java project**



1.3 Name the project **Demo48,** uncheck **Create a module info dot Java file**, and click **Finish**

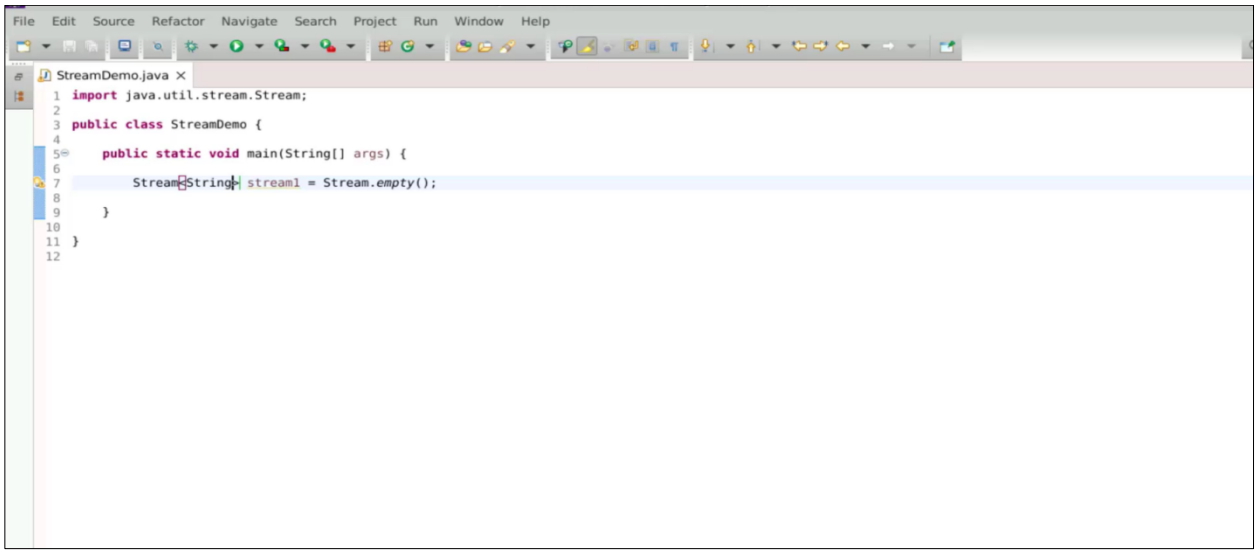


1.4 In the **src** folder, right-click and create a new class. Name this class **StreamDemo**. Select the main method and then click **Finish**



Step 2: Create one of the stream objects

2.1 First, understand how to create an empty stream object. Import the **Stream** API from the java.util package. The Stream interface provides a method called **empty** that returns an empty sequential stream. Enter **Stream.empty()** to create an empty stream. Then assign it to a variable called stream1 like this: **Stream<String> stream1 = Stream.empty()**



2.2 Now, create an **ArrayList** of strings called emails and add some email addresses to it. For example**, john@example.com** and a few more emails

```
import java.util.ArrayList;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

    }

}
```

2.3 To create a stream from the **ArrayList**, use the **stream()** method. Enter **emails.stream()** to create a stream from the ArrayList. Assign it to a variable called **stream2** like this: **Stream<String> stream2 = emails.stream()**. Alternatively, you can use the **Arrays.asList()** method to create a list from an array and then create a stream directly from it

```
import java.util.ArrayList;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

    }

}
```

2.4 For example, use the **Arrays.asList()** method to create a list of strings containing a few email addresses. Add **fionna@example.com** to the list. Then, call the **stream()** method on the list to create a stream. Assign it to a variable called **stream3** like this: **Stream<String> stream3 = Arrays.asList(emails).stream()**

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

    }

}
```

2.5 In this case, you are just getting references to the streams without actually processing them. It helps in understanding how to obtain the stream references

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

    }

}
```

2.6 Additionally, if you have an array of strings, you can create a stream from it. Create a string array to store email addresses

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};

    }

}
```

2.7 To create a stream from the array, use the **stream()** method. Write **Stream<String> stream4 = Arrays.stream(emailsArray)** to create a stream from the array. Assign it to a variable called **stream4**

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

    }

}
```

2.8 Alternatively, you can use the static method, **Stream.of()** to directly create a stream by passing the values. **Write Stream<String> stream5 = Stream.of(emails)** to create a stream from the **emails** array. Assign it to a variable called **stream5**

```
File Edit Source Refactor Navigate Search Project Run Window Help
StreamDemo.java ×
1  import java.util.ArrayList;
2  import java.util.Arrays;
3  import java.util.stream.Stream;
4
5  public class StreamDemo {
6
7      public static void main(String[] args) {
8
9          Stream<String> stream1 = Stream.empty();
10
11         ArrayList<String> emails = new ArrayList<String>();
12         emails.add("john@example.com");
13         emails.add("harry@example.com");
14         emails.add("leo@example.com");
15         emails.add("kim@example.com");
16         emails.add("george@example.com");
17
18         Stream<String> stream2 = emails.stream();
19
20         Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();
21
22         String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
23         Stream<String> stream4 = Arrays.stream(emailsArray);
24
25         Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");
26
27     }
28
29 }
30
```

2.9 Another option is to use the stream builder API. Define a stream of strings and use the static method **builder()** from the **Stream** class. Write **Stream<String> stream6 = Stream.<String>builder()** to create a stream builder for strings. Assign it to a variable called **stream6**

```
File Edit Source Refactor Navigate Search Project Run Window Help
*StreamDemo.java ×
1  import java.util.ArrayList;
2  import java.util.Arrays;
3  import java.util.stream.Stream;
4  import java.util.stream.Stream.Builder;
5
6  public class StreamDemo {
7
8      public static void main(String[] args) {
9
10         Stream<String> stream1 = Stream.empty();
11
12         ArrayList<String> emails = new ArrayList<String>();
13         emails.add("john@example.com");
14         emails.add("harry@example.com");
15         emails.add("leo@example.com");
16         emails.add("kim@example.com");
17         emails.add("george@example.com");
18
19         Stream<String> stream2 = emails.stream();
20
21         Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();
22
23         String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
24         Stream<String> stream4 = Arrays.stream(emailsArray);
25
26         Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");
27
28         Builder<String> builder = Stream.builder();
29
30     }
31
```

2.10 On the stream builder, you can add the email addresses using the **add()** method. Finally, call the **build()** method on the builder to create the stream. Write **Stream<String> stream7 = stream6.add(emails).build()** to create a stream from the stream builder. Assign it to a variable called **stream7**

```
File  Edit  Source  Refactor  Navigate  Search  Project  Run  Window  Help
StreamDemo.java ×
1  import java.util.ArrayList;
2  import java.util.Arrays;
3  import java.util.stream.Stream;
4  import java.util.stream.Stream.Builder;
5
6  public class StreamDemo {
7
8      public static void main(String[] args) {
9
10         Stream<String> stream1 = Stream.empty();
11
12         ArrayList<String> emails = new ArrayList<String>();
13         emails.add("john@example.com");
14         emails.add("harry@example.com");
15         emails.add("leo@example.com");
16         emails.add("kim@example.com");
17         emails.add("george@example.com");
18
19         Stream<String> stream2 = emails.stream();
20
21         Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();
22
23         String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
24         Stream<String> stream4 = Arrays.stream(emailsArray);
25
26         Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");
27
28         Builder<String> builder = Stream.builder();
29         builder.add("john@example.com");
30         builder.add("harry@example.com");
31         builder.add("leo@example.com");
32         builder.add("kim@example.com");
33         builder.add("george@example.com");
34
35         Stream<String> stream6 = builder.build();
36
37     }
38
```

Steps 3: Create a stream using various approaches

3.1 There are a couple more ways to create a stream, explore the first method called **generate**. Write **Stream.generate()** and provide a lambda expression inside it. For example, generate the number 101. Add **.limit(30)** to limit the stream to a maximum of 30 elements. In this case, the stream, **stream7**, will have the data type of Integer

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;
import java.util.stream.Stream.Builder;

public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");

        Builder<String> builder = Stream.builder();
        builder.add("john@example.com");
        builder.add("harry@example.com");
        builder.add("leo@example.com");
        builder.add("kim@example.com");
        builder.add("george@example.com");

        Stream<String> stream6 = builder.build();

        Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);
```

3.2 Now working with the **iterate** method. Create another stream called **stream8**. Instead of using the **generate** method, use the **iterate** method. Start with the number 100 and define a function that takes a number as input and adds 2 to it. Add limit**(20)** to limit the stream to a maximum of 20 elements

```java
public class StreamDemo {

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");

        Builder<String> builder = Stream.builder();
        builder.add("john@example.com");
        builder.add("harry@example.com");
        builder.add("leo@example.com");
        builder.add("kim@example.com");
        builder.add("george@example.com");

        Stream<String> stream6 = builder.build();

        Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);

        Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);
    }
```

Step 4: Iterate the streams and execute the code

4.1 Now, iterate through streams that contain data. Start by iterating through **stream2**. Write **stream2.forEach()** and provide a function as input. Inside the function, choose a name **element** and execute the **System.out.println()** method to print the element

```
*StreamDemo.java
6  public class StreamDemo {
7
8      public static void main(String[] args) {
9
10         Stream<String> stream1 = Stream.empty();
11
12         ArrayList<String> emails = new ArrayList<String>();
13         emails.add("john@example.com");
14         emails.add("harry@example.com");
15         emails.add("leo@example.com");
16         emails.add("kim@example.com");
17         emails.add("george@example.com");
18
19         Stream<String> stream2 = emails.stream();
20
21         Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();
22
23         String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
24         Stream<String> stream4 = Arrays.stream(emailsArray);
25
26         Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");
27
28         Builder<String> builder = Stream.builder();
29         builder.add("john@example.com");
30         builder.add("harry@example.com");
31         builder.add("leo@example.com");
32         builder.add("kim@example.com");
33         builder.add("george@example.com");
34
35         Stream<String> stream6 = builder.build();
36
37         Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);
38
39         Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);
40
41         stream2.forEach((element)->System.out.println(element));
42     }
43
```

4.2 Run the code and observe the iteration in action. You will see the elements of the stream being printed

```
StreamDemo.java   Run StreamDemo (already running)
6  public class StreamDemo {
7
8      public static void main(String[] args) {
9
10         Stream<String> stream1 = Stream.empty();
11
12         ArrayList<String> emails = new ArrayList<String>();
13         emails.add("john@example.com");
14         emails.add("harry@example.com");
15         emails.add("leo@example.com");
16         emails.add("kim@example.com");
17         emails.add("george@example.com");
18
19         Stream<String> stream2 = emails.stream();
20
21         Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fi
22
23         String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fion
24         Stream<String> stream4 = Arrays.stream(emailsArray);
25
26         Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna(
27
28         Builder<String> builder = Stream.builder();
29         builder.add("john@example.com");
30         builder.add("harry@example.com");
31         builder.add("leo@example.com");
32         builder.add("kim@example.com");
33         builder.add("george@example.com");
34
35         Stream<String> stream6 = builder.build();
36

Problems   Javadoc   Declaration   Console
<terminated> StreamDemo [Java Application] /usr/eclipse/plugins/org.eclips
john@example.com
harry@example.com
leo@example.com
kim@example.com
george@example.com
```

4.3 Now, iterate through the stream again and rerun the code. However, this time you will encounter an exception

```
class StreamDemo {

lic static void main(String[] args) {

 Stream<String> stream1 = Stream.empty();

 ArrayList<String> emails = new ArrayList<String>();
 emails.add("john@example.com");
 emails.add("harry@example.com");
 emails.add("leo@example.com");
 emails.add("kim@example.com");
 emails.add("george@example.com");

 Stream<String> stream2 = emails.stream();

 Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fi

 String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fion
 Stream<String> stream4 = Arrays.stream(emailsArray);

 Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna

 Builder<String> builder = Stream.builder();
 builder.add("john@example.com");
 builder.add("harry@example.com");
 builder.add("leo@example.com");
 builder.add("kim@example.com");
 builder.add("george@example.com");

 Stream<String> stream6 = builder.build();

 Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);

 Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);

 stream2.forEach((element)->System.out.println(element));
 stream2.forEach((element)->System.out.println(element));
```

```
<terminated> StreamDemo [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk
john@example.com
harry@example.com
leo@example.com
kim@example.com
george@example.com
Exception in thread "main" java.lang.IllegalStateException: stream has alr
        at java.base/java.util.stream.AbstractPipeline.sourceStageSplitera
        at java.base/java.util.stream.ReferencePipeline$Head.forEach(Refer
        at StreamDemo.main(StreamDemo.java:42)
```

4.4 This exception occurs because once a stream has been processed, it is considered closed. You cannot iterate through the same stream again

```
<terminated> StreamDemo [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.1.v20211116-1657/jre/bin/java (Feb 2, 2022, 5:34:24 PM – 5:34:24 PM)
john@example.com
harry@example.com
leo@example.com
kim@example.com
george@example.com
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
        at java.base/java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:279)
        at java.base/java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:762)
        at StreamDemo.main(StreamDemo.java:42)
```

4.5 Next, create a static method called **iterateStream** that takes a stream of type String as a parameter. Copy the stream definition and pass it as an argument to the method

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;
import java.util.stream.Stream.Builder;

public class StreamDemo {

    static void iterateStream(Stream<String> stream) {
        stream.forEach((element)->System.out.println(element));
    }

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");
```

4.6 Execute the **iterateStream** method and pass **stream3** as the argument

```java
        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");

        Builder<String> builder = Stream.builder();
        builder.add("john@example.com");
        builder.add("harry@example.com");
        builder.add("leo@example.com");
        builder.add("kim@example.com");
        builder.add("george@example.com");

        Stream<String> stream6 = builder.build();

        Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);

        Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);

        stream2.forEach((element)->System.out.println(element));
        //stream2.forEach((element)->System.out.println(element)); // exception

        iterateStream(stream3);
    }
```

4.7 Before starting the iteration, add an empty **System.out.println()** statement. Run the code, and you will see the elements **John**, **Jenny**, and **Fionna** being printed

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;
import java.util.stream.Stream.Builder;

public class StreamDemo {

    static void iterateStream(Stream<String> stream) {
        System.out.println();
        stream.forEach((element)->System.out.println(element));
    }

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");
```

```
<terminated> StreamDemo [Java Application] /usr/eclipse/plugins/org.eclips
john@example.com
harry@example.com
leo@example.com
kim@example.com
george@example.com

john@example.com
jennie@example.com
fionna@example.com
```

4.8 Similarly, you can iterate through stream6 which contains email addresses

```
        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fi

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fion
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna(

        Builder<String> builder = Stream.builder();
        builder.add("john@example.com");
        builder.add("harry@example.com");
        builder.add("leo@example.com");
        builder.add("kim@example.com");
        builder.add("george@example.com");

        Stream<String> stream6 = builder.build();

        Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);

        Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);

        stream2.forEach((element)->System.out.println(element));
        //stream2.forEach((element)->System.out.println(element)); // exception

        iterateStream(stream6);
    }

}
```

```
<terminated> StreamDemo [Java Application] /usr/eclipse/plugins/org.eclips
john@example.com
harry@example.com
leo@example.com
kim@example.com
george@example.com

john@example.com
harry@example.com
leo@example.com
kim@example.com
george@example.com
```

4.9 If you want to iterate through integers, make sure to change the type to Integer

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Stream;
import java.util.stream.Stream.Builder;

public class StreamDemo {

    static void iterateStream(Stream<Integer> stream) {
        System.out.println();
        stream.forEach((element)->System.out.println(element));
    }

    public static void main(String[] args) {

        Stream<String> stream1 = Stream.empty();

        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");
```

4.10 Pass **stream7** as the argument to the **iterateStream** method. Run the code, and you will see **101** appearing 30 times

```java
        ArrayList<String> emails = new ArrayList<String>();
        emails.add("john@example.com");
        emails.add("harry@example.com");
        emails.add("leo@example.com");
        emails.add("kim@example.com");
        emails.add("george@example.com");

        Stream<String> stream2 = emails.stream();

        Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fionna@example.com").stream();

        String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fionna@example.com"};
        Stream<String> stream4 = Arrays.stream(emailsArray);

        Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna@example.com");

        Builder<String> builder = Stream.builder();
        builder.add("john@example.com");
        builder.add("harry@example.com");
        builder.add("leo@example.com");
        builder.add("kim@example.com");
        builder.add("george@example.com");

        Stream<String> stream6 = builder.build();

        Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);

        Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);

        stream2.forEach((element)->System.out.println(element));
        //stream2.forEach((element)->System.out.println(element)); // exception

        iterateStream(stream7);
    }

}
```

4.11 You can modify the limit to display **101** five times. Run the code, and you will see the desired output

```
ArrayList<String> emails = new ArrayList<String>();
emails.add("john@example.com");
emails.add("harry@example.com");
emails.add("leo@example.com");
emails.add("kim@example.com");
emails.add("george@example.com");

Stream<String> stream2 = emails.stream();

Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fi

String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fion
Stream<String> stream4 = Arrays.stream(emailsArray);

Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna

Builder<String> builder = Stream.builder();
builder.add("john@example.com");
builder.add("harry@example.com");
builder.add("leo@example.com");
builder.add("kim@example.com");
builder.add("george@example.com");

Stream<String> stream6 = builder.build();

Stream<Integer> stream7 = Stream.generate(()-> 101).limit(30);

Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);

stream2.forEach((element)->System.out.println(element));
```

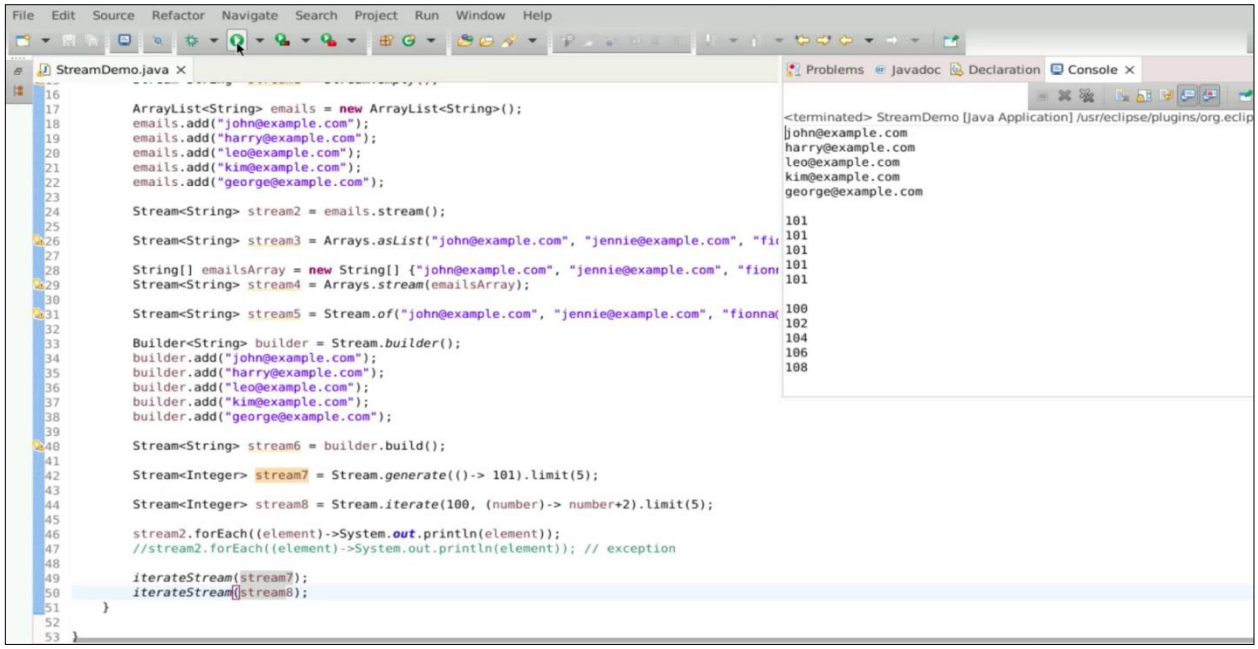
```
<terminated> StreamDemo [Java Application] /usr/eclipse/plugins/org.eclips
101
101
101
101
101
101
101
101
101
101
101
101
101
101
101
101
101
101
```

4.12 Now let us iterate through "stream8". Pass "stream8" as the argument to the **iterateStream** method. Run the code, and you will observe that the number increments by two each time

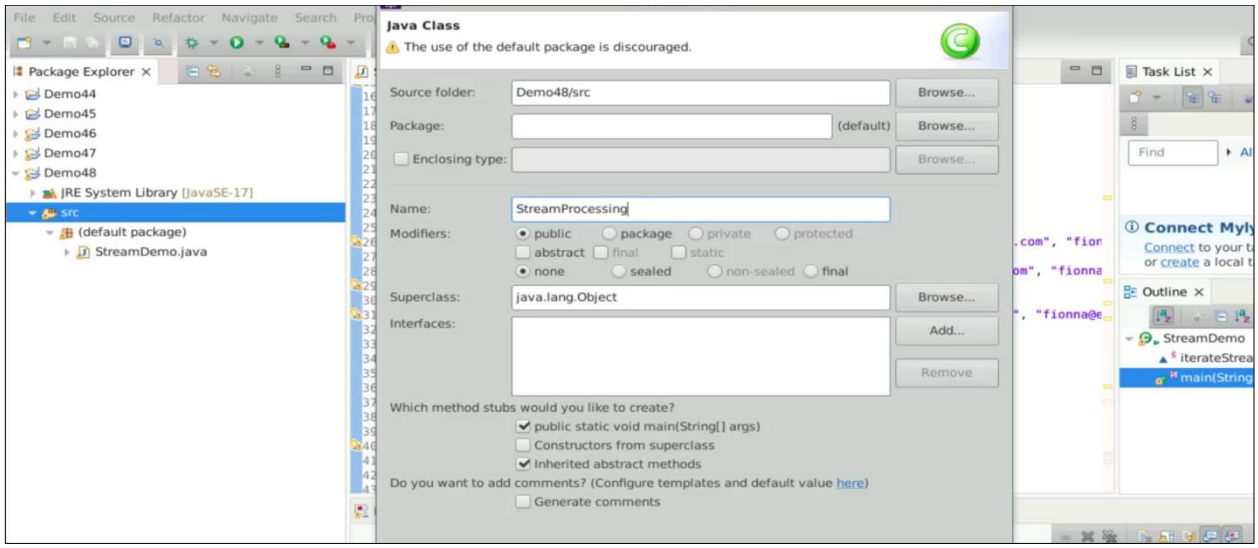
```
ArrayList<String> emails = new ArrayList<String>();
emails.add("john@example.com");
emails.add("harry@example.com");
emails.add("leo@example.com");
emails.add("kim@example.com");
emails.add("george@example.com");

Stream<String> stream2 = emails.stream();

Stream<String> stream3 = Arrays.asList("john@example.com", "jennie@example.com", "fi

String[] emailsArray = new String[] {"john@example.com", "jennie@example.com", "fion
Stream<String> stream4 = Arrays.stream(emailsArray);

Stream<String> stream5 = Stream.of("john@example.com", "jennie@example.com", "fionna

Builder<String> builder = Stream.builder();
builder.add("john@example.com");
builder.add("harry@example.com");
builder.add("leo@example.com");
builder.add("kim@example.com");
builder.add("george@example.com");

Stream<String> stream6 = builder.build();

Stream<Integer> stream7 = Stream.generate(()-> 101).limit(5);

Stream<Integer> stream8 = Stream.iterate(100, (number)-> number+2).limit(20);

stream2.forEach((element)->System.out.println(element));
```

```
<terminated> StreamDemo [Java Application] /usr/eclipse/plugins/org.eclips
john@example.com
harry@example.com
leo@example.com
kim@example.com
george@example.com

101
101
101
101
101
```

4.13 While using iteration, the function retrieves the latest value of the number, not necessarily adding two to the previous value



4.14 Create a new class named **StreamProcessing** with a main method to continue with further steps



Step 5: Create a stream of integer numbers and execute the code

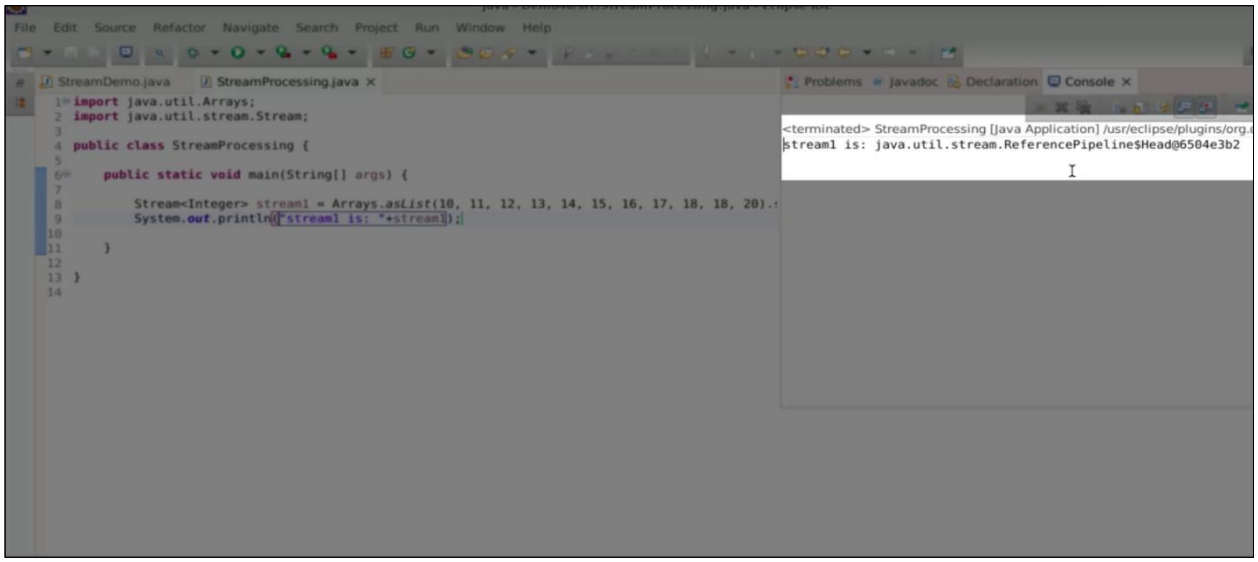
5.1 Now, create a stream of integer numbers using the Arrays.asList() method. Specify the numbers, such as **10, 11, 12, 13, 14, 15**, and so on. Capture this as a stream by invoking the **stream()** method. Create a stream of type **Integer** called **stream1** and import the **Stream** class from the **java.util** package

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();

    }

}
```

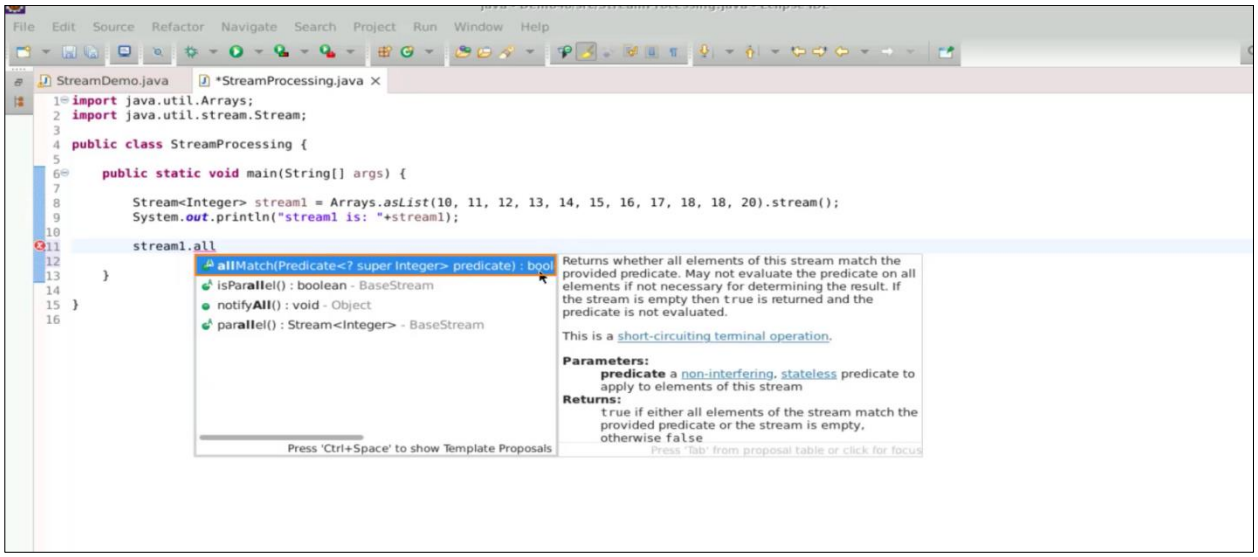
5.2 Execute a method called **allMatch()** on the stream. Print the stream using **System.out.println()** to verify the reference pipeline object

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

    }

}
```

5.3 When you run the code, you will notice that the printed output is an object reference. Unlike collections, streams do not directly display their data



5.4 Next, use the **allMatch()** method on **stream1**. It returns a Boolean value and takes a predicate as input. The predicate function determines whether the result is true or false



5.5 Create a Boolean variable to store the result of the **allMatch()** method. Pass a Lambda expression as the predicate function. The input to the Lambda expression is the number, and the expression to test is whether all numbers are greater than 15

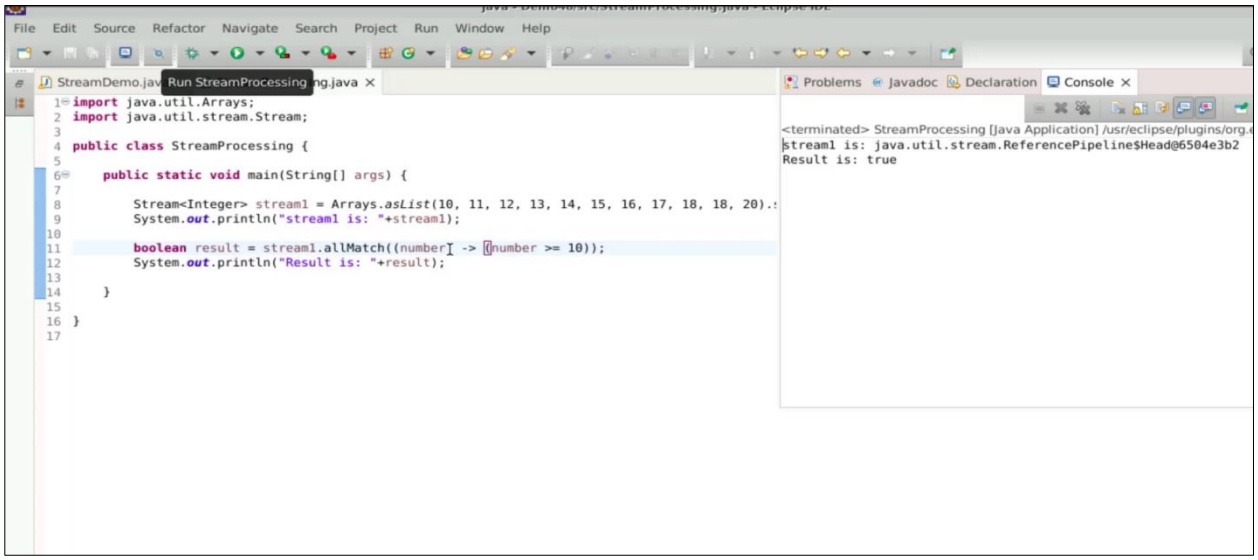
```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number > 15));

    }

}
```

5.6 Print the result using **System.out.println()**. In this case, you are testing if all the numbers are greater than 15. Since there are numbers less than 15, the result will be false

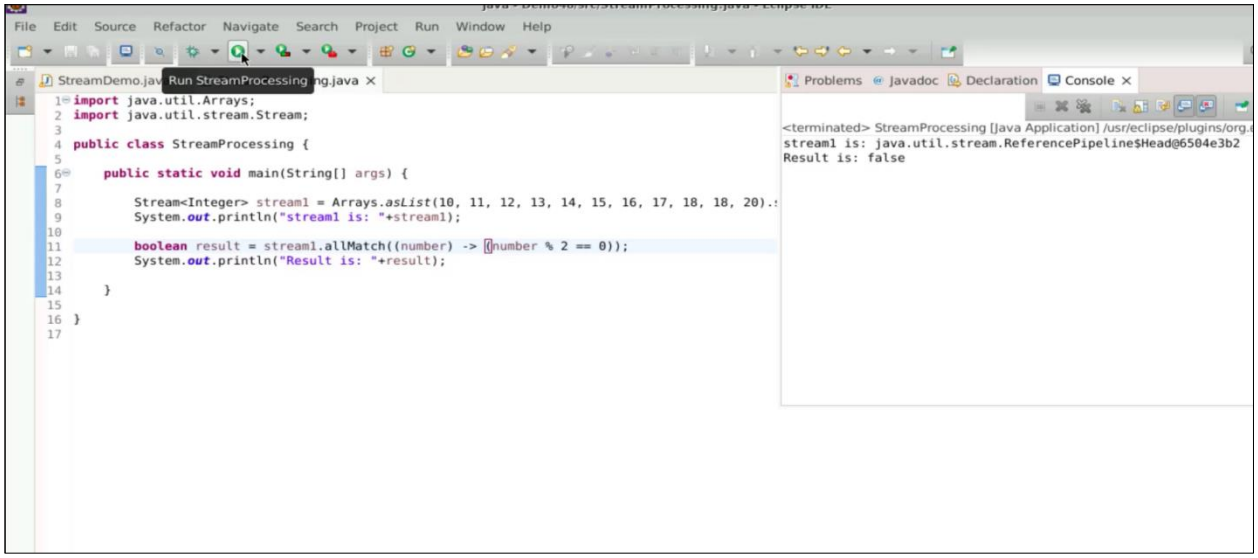
```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number > 15));
        System.out.println("Result is: "+result);

    }

}
```

```
<terminated> StreamProcessing [Java Application] /usr/eclipse/plugins/org.
stream1 is: java.util.stream.ReferencePipeline$Head@6504e3b2
Result is: false
```

5.7 Modify the condition to check if all numbers are greater than or equal to 10. When you run the code, it should display true



5.8 You can also apply different conditions, such as checking if all numbers are even using the condition **number % 2 == 0**. When you run the code, it displays false. The **allMatch()** method tests the specified condition on all elements in the stream



5.9 If you try to iterate through the stream using **forEach()** after processing it, you will encounter an error. Once a stream has been processed, it cannot be processed again

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).s
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        stream1.forEach((number)-> System.out.println(number));

    }

}
```

```
<terminated> StreamProcessing [Java Application] /usr/eclipse/plugins/org.e
stream1 is: java.util.stream.ReferencePipeline$Head@6504e3b2
Result is: false
Exception in thread "main" java.lang.IllegalStateException: st
        at java.base/java.util.stream.AbstractPipeline.sourceS
        at java.base/java.util.stream.ReferencePipeline$Head.f
        at StreamProcessing.main(StreamProcessing.java:14)
```

5.10 Create another stream called "**stream2**" with numbers 10, 20, 30, 40, and 50. Use the **reduce()** method on this stream. **reduce()** takes a binary operation as input and returns an optional

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        stream2.red

    }

}
```

```
reduce(BinaryOperator<Integer> accumulator) : Optic
reduce(Integer identity, BinaryOperator<Integer> acc
reduce(U identity, BiFunction<U,? super Integer,U> ac
forEachOrdered(Consumer<? super Integer> action) :
unordered() : Stream<Integer> - BaseStream
Press 'Ctrl+Space' to show Template Proposals
```

Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any. This is equivalent to:

```
boolean foundAny = false;
    T result = null;
    for (T element : this stream) {
        if (!foundAny) {
            foundAny = true;
            result = element;
        }
        else
            result = accumulator.apply(result
    }
```

Press 'Tab' from proposal table or click for focus

5.11 Pass a function to **reduce()** that takes two numbers as input **(num1, num2)** and returns their sum

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        stream2.reduce((num1, num2) -> num1+num2);

    }

}
```

5.12 Assign the result of **stream2.reduce()** to an optional variable called **optionalResult**. Print the **optionalResult** to see how the stream has been processed. When you run the code, the optional value will be **150**

```
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult);

    }

}
```

```
<terminated> StreamProcessing [Java Application] /usr/eclipse/plugins/org.e
stream1 is: java.util.stream.ReferencePipeline$Head@6504e3b2
Result is: false
Optional[150]
```

5.13 Use the **get()** method on the optional to retrieve the value. When you run the code, it correctly displays **150**. The **reduce()** method is typically used to reduce all elements of a stream to a single element

```
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).s
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

    }

}
```

```
<terminated> StreamProcessing [Java Application] /usr/eclipse/plugins/org.
stream1 is: java.util.stream.ReferencePipeline$Head@6504e3b2
Result is: false
150
```

Step 6: Generate another stream and re-execute the code

6.1 Now, generate another stream called **productPrices**. Write the product prices as **1200, 3500, 2400, 1600, and 2100**

```
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

        Stream<Integer> streamProductPrices = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();

    }

}
```

6.2 Now, we want to filter the stream. Use the **filter()** method on the **productPrices** stream. **filter()** takes a predicate function as input, where you can specify your criteria for filtering the data. In this case, use a simple condition: **element >=2000**

```java
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

        Stream<Integer> streamProductPrices = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        streamProductPrices.filter((element) -> element >=2000);

    }

}
```

6.3 The filter function will return a new stream containing the filtered elements. Assign this filtered stream to a new variable called **filteredProductPrices**

```java
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

        Stream<Integer> streamProductPrices = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        Stream<Integer> filteredProductPrices = streamProductPrices.filter((element) -> element >=2000);

    }

}
```

6.4 Next, iterate through the filtered stream using the **forEach()** method. Take the price as input in the Lambda expression and print the price. The input variable can be named as per your choice. When you run the code, it will display the prices **3500**, **2400**, and **2100**

```java
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

        System.out.println();

        Stream<Integer> streamProductPrices = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        Stream<Integer> filteredProductPrices = streamProductPrices.filter((element) -> element >=2000);
        filteredProductPrices.forEach((price) -> System.out.println(price));

    }

}
```

```
<terminated> StreamProcessing [Java Application] /usr/eclipse/plugins/org.e
stream1 is: java.util.stream.ReferencePipeline$Head@6504e3b2
Result is: false
150

3500
2400
2100
```

6.5 Create another stream called **productPrices1** with the same elements

```java
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

        System.out.println();

        Stream<Integer> streamProductPrices = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        Stream<Integer> filteredProductPrices = streamProductPrices.filter((element) -> element >= 2000);
        filteredProductPrices.forEach((price) -> System.out.println(price));

        Stream<Integer> streamProductPrices1 = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();


    }
```

6.6 To apply the discount, use the **map()** method on the **productPrices1** stream. Inside the **map()** function, specify the input variable (price in this case). Change the variable name to **mappedProductPrices**. Use the **forEach()** method on **mappedProductPrices** and take the price as input in the Lambda expression

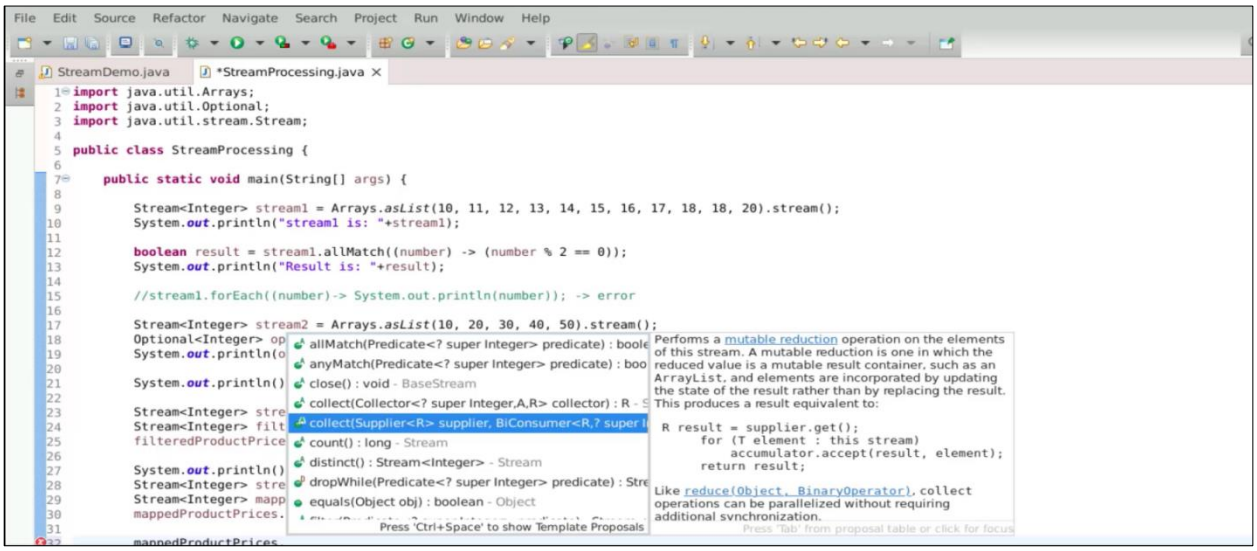
```
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).stream();
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

        System.out.println();

        Stream<Integer> streamProductPrices = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        Stream<Integer> filteredProductPrices = streamProductPrices.filter((element) -> element >= 2000);
        filteredProductPrices.forEach((price) -> System.out.println(price));

        System.out.println();
        Stream<Integer> streamProductPrices1 = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        Stream<Integer> mappedProductPrices = streamProductPrices1.map((price)-> price-200);
        mappedProductPrices.forEach((price) -> System.out.println(price));
```

6.7 Run the code, and you will see that 1200 becomes **1000**, 3500 becomes **3300**, and so on. It's important to note that streams are not limited to just integers or strings; they can also handle user-defined objects

```
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamProcessing {

    public static void main(String[] args) {

        Stream<Integer> stream1 = Arrays.asList(10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 20).s
        System.out.println("stream1 is: "+stream1);

        boolean result = stream1.allMatch((number) -> (number % 2 == 0));
        System.out.println("Result is: "+result);

        //stream1.forEach((number)-> System.out.println(number)); -> error

        Stream<Integer> stream2 = Arrays.asList(10, 20, 30, 40, 50).stream();
        Optional<Integer> optionalResult = stream2.reduce((num1, num2) -> num1+num2);
        System.out.println(optionalResult.get());

        System.out.println();

        Stream<Integer> streamProductPrices = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        Stream<Integer> filteredProductPrices = streamProductPrices.filter((element) -> element >= 2000);
        filteredProductPrices.forEach((price) -> System.out.println(price));

        System.out.println();
        Stream<Integer> streamProductPrices1 = Arrays.asList(1200, 3500, 2400, 1600, 2100).stream();
        Stream<Integer> mappedProductPrices = streamProductPrices1.map((price)-> price-200);
        mappedProductPrices.forEach((price) -> System.out.println(price));
```

```
<terminated> StreamProcessing [Java Application] /usr/eclipse/plugins/org.e
stream1 is: java.util.stream.ReferencePipeline$Head@6504e3b2
Result is: false
150

3500
2400
2100

1000
3300
2200
1400
1900
```

6.8 Apart from the methods discussed so far (allMatch, reduce, filter, map), there are several other methods available to work with streams. For example, you can use the count() method to get the number of elements in the stream or the **collect()** method to collect the elements in a specific way. Explore the documentation to discover more options



By following these steps, you have successfully implemented streams in Java to perform functional-style operations on collections and improve code readability and efficiency.