

Java 8 Features

Github link for all questions: https://github.com/hema-alapati/Hemalatha_RG-tasks and inside **branch:** feature-java8

1. List the features of Java 8

Java 8 introduced some of the most powerful features in the language's evolution. Some of the key highlights are as follows:

Lambda Expressions – Enables functional-style programming by allowing you to write concise, inline functions.

Functional Interfaces – Interfaces with exactly one abstract method, often used as the target type for lambda expressions.

Stream API – Provides powerful data processing capabilities using pipelines.

Date and Time API (java.time package) – A modern, immutable, and thread-safe approach to date and time.

Default and Static Methods in Interfaces – Interfaces can now have method implementations.

Optional Class – Helps avoid NullPointerException by representing optional values.

Method References – Shorthand notation for calling methods with lambdas.

Nashorn JavaScript Engine – For running JavaScript code on the JVM (mainly deprecated now)

New Collectors in Stream API – Such as `toList()`, `joining()`, `groupingBy()`.

Parallel Streams – Enables easy multithreaded stream processing.

2. What is a Lambda Expression, and why do we use them? Explain with a coding example and share the output screenshot.

Lambda expressions are a shorthand way to create anonymous functions in Java. They're mainly used to make code more readable and concise, especially when working with functional interfaces or APIs like Stream.

Example:

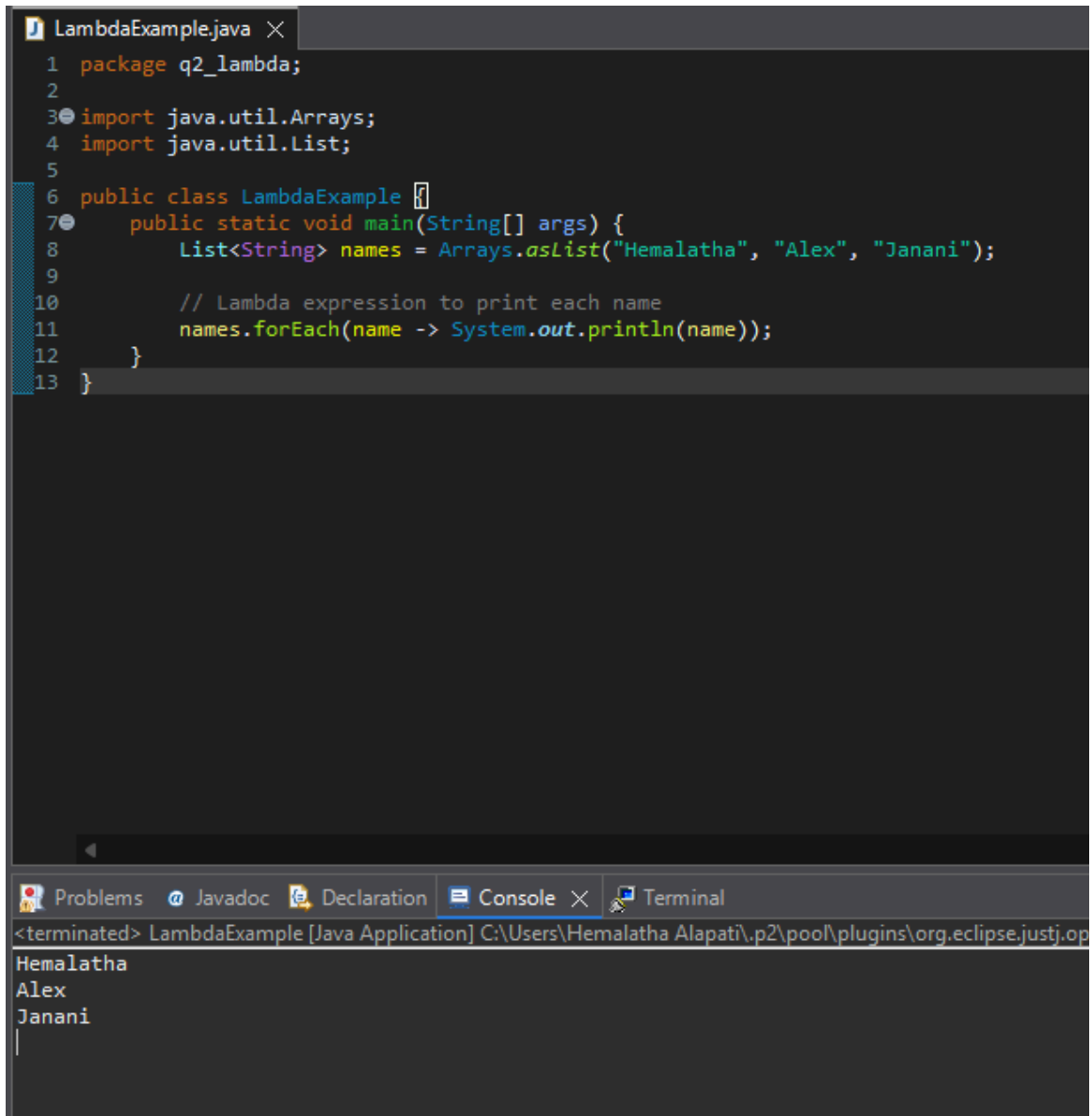
```
package q2_lambda;

import java.util.Arrays;

import java.util.List;
```

```
public class LambdaExample {  
    public static void main(String[] args) {  
        List<String> names = Arrays.asList("Hemalatha", "Alex", "Janani");  
        // Lambda expression to print each name  
        names.forEach(name -> System.out.println(name));  
    }  
}
```

Output:



```
1 package q2_lambda;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 public class LambdaExample {
7     public static void main(String[] args) {
8         List<String> names = Arrays.asList("Hemalatha", "Alex", "Janani");
9
10        // Lambda expression to print each name
11        names.forEach(name -> System.out.println(name));
12    }
13 }
```

The screenshot shows the Eclipse IDE interface. The top editor window displays the source code for `LambdaExample.java`. The code defines a package `q2_lambda`, imports `java.util.Arrays` and `java.util.List`, and contains a `main` method. Inside `main`, a `List<String>` named `names` is created using `Arrays.asList` with the values "Hemalatha", "Alex", and "Janani". A lambda expression `forEach` is used to iterate over the list and print each name to the console. The bottom of the IDE shows the `Console` tab, which displays the output of the program: `Hemalatha`, `Alex`, and `Janani`, each on a new line.

3. What is optional, and what is it best used for? **Explain with a coding example and share the output screenshot.**

Optional<T> is a container object that may or may not contain a non-null value. It's a better alternative to returning null and helps in avoiding NullPointerException.\

Example:

```
package q3_optional;

import java.util.Optional;

public class OptionalExample {

    public static void main(String[] args) {

        String name = "Hemalatha";

        Optional<String> optionalName = Optional.ofNullable(name);

        optionalName.ifPresentOrElse(

            val -> System.out.println("Name is: " + val),

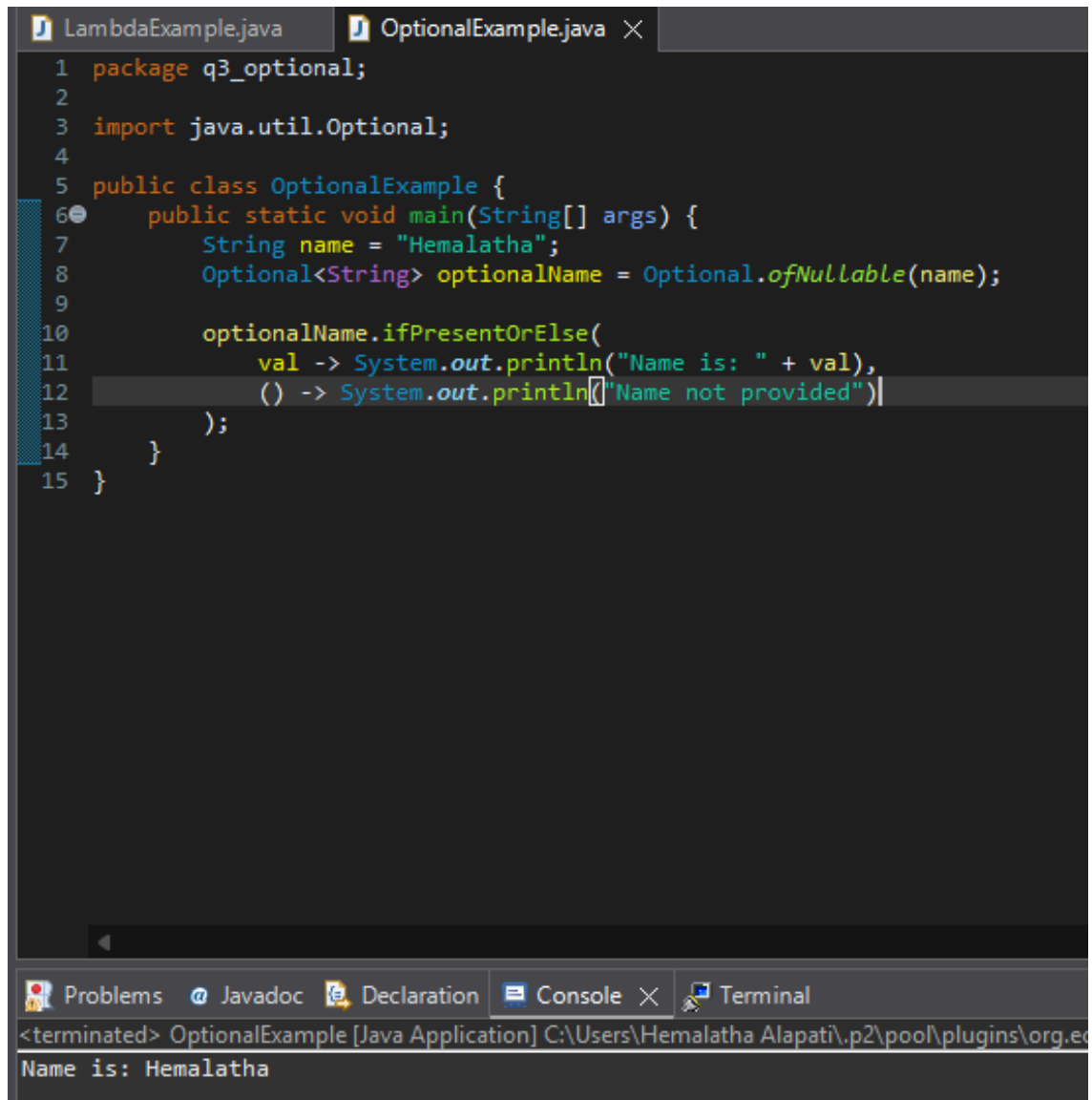
            () -> System.out.println("Name not provided")

        );

    }

}
```

Output:



The screenshot shows an IDE with two tabs: 'LambdaExample.java' and 'OptionalExample.java'. The 'OptionalExample.java' tab is active, displaying the following Java code:

```
1 package q3_optional;
2
3 import java.util.Optional;
4
5 public class OptionalExample {
6     public static void main(String[] args) {
7         String name = "Hemalatha";
8         Optional<String> optionalName = Optional.ofNullable(name);
9
10        optionalName.ifPresentOrElse(
11            val -> System.out.println("Name is: " + val),
12            () -> System.out.println("Name not provided")
13        );
14    }
15 }
```

Below the code editor, there is a toolbar with icons for 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Terminal'. The 'Console' tab is selected, showing the output of the program:

```
<terminated> OptionalExample [Java Application] C:\Users\Hemalatha Alapati\p2\pool\plugins\org.e
Name is: Hemalatha
```

4. What is a functional interface? List some examples of predefined functional interfaces.

Functional Interface:

A **functional interface** in Java is an interface that contains exactly one abstract method. These are meant to be used with lambda expressions or method references.

Examples of predefined functional interfaces:

Interface	Method	Description
Predicate<T>	test(T t)	Returns true/false based on input
Function<T, R>	apply(T t)	Converts type T to type R
Consumer<T>	accept(T t)	Consumes input without return
Supplier<T>	get()	Supplies a value of type T

5. How are functional interfaces and Lambda Expressions related?

Lambda expressions are possible **only because of functional interfaces**. When a method expects a functional interface, you can pass a lambda expression instead of writing a full anonymous class.

Example:

```
package q5_FI;

import java.util.function.Consumer;

public class FunctionalRelation {

    public static void main(String[] args) {

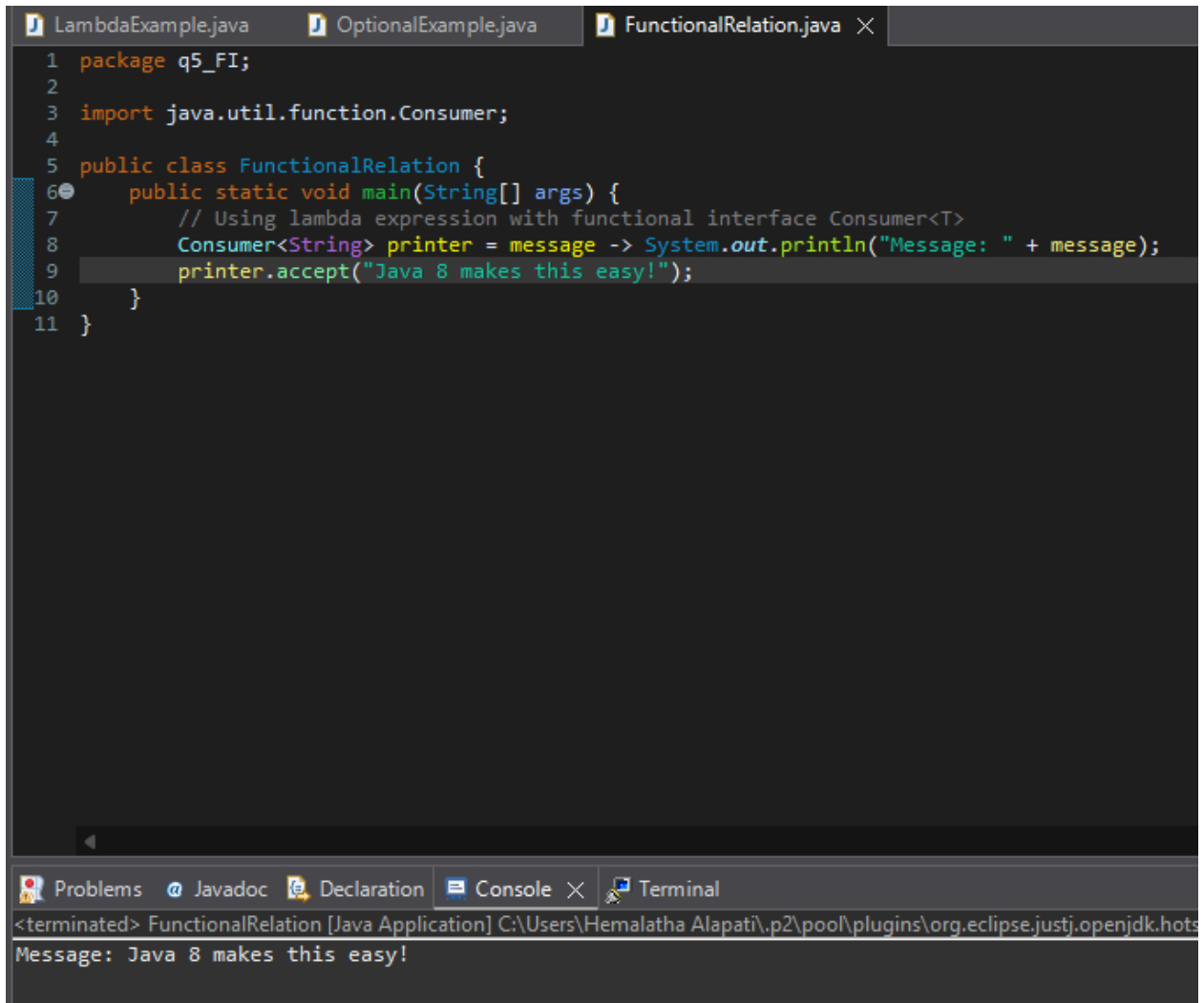
        // Using lambda expression with functional interface Consumer<T>

        Consumer<String> printer = message -> System.out.println("Message: " + message);

        printer.accept("Java 8 makes this easy!");
```

```
}  
  
}
```

Output:



The screenshot shows an IDE with three tabs: LambdaExample.java, OptionalExample.java, and FunctionalRelation.java. The FunctionalRelation.java tab is active, displaying the following code:

```
1 package q5_FI;  
2  
3 import java.util.function.Consumer;  
4  
5 public class FunctionalRelation {  
6     public static void main(String[] args) {  
7         // Using lambda expression with functional interface Consumer<T>  
8         Consumer<String> printer = message -> System.out.println("Message: " + message);  
9         printer.accept("Java 8 makes this easy!");  
10    }  
11 }
```

At the bottom, the Console tab shows the output:

```
<terminated> FunctionalRelation [Java Application] C:\Users\Hemalatha Alapati\p2\pool\plugins\org.eclipse.justj.openjdk.hot  
Message: Java 8 makes this easy!
```

6. List some Java 8 Date and Time API's. **How will you get the current date and time using Java 8 Date and Time API? Write the implementation and share the output screenshot.**

Java 8 introduced a new date/time API under java.time package. It's immutable, thread-safe, and inspired by Joda-Time.

Common Classes:

- LocalDate – Only date
- LocalTime – Only time
- LocalDateTime – Date and time
- ZonedDateTime – Date, time, and time zone
- Period, Duration – For difference between dates/times
- DateTimeFormatter – For formatting/parsing

Implementation:

```
package q6_DateTime;

import java.time.LocalDateTime;

public class DateTimeNow {

    public static void main(String[] args) {

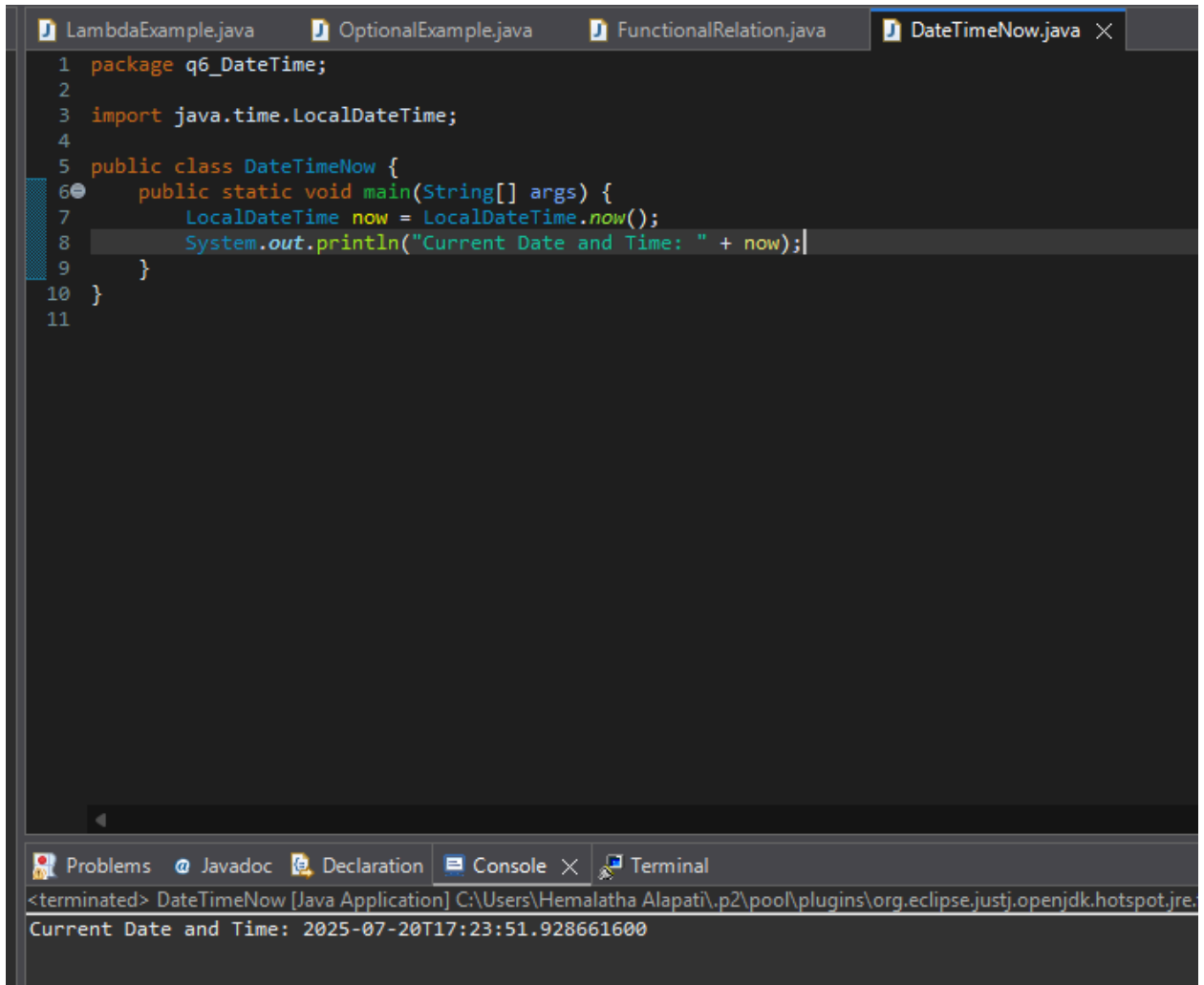
        LocalDateTime now = LocalDateTime.now();

        System.out.println("Current Date and Time: " + now);

    }

}
```


Output:



```
1 package q6_DateTime;  
2  
3 import java.time.LocalDateTime;  
4  
5 public class DateTimeNow {  
6     public static void main(String[] args) {  
7         LocalDateTime now = LocalDateTime.now();  
8         System.out.println("Current Date and Time: " + now);  
9     }  
10 }  
11
```

Problems Javadoc Declaration Console × Terminal

<terminated> DateTimeNow [Java Application] C:\Users\Hemalatha Alapati\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre;
Current Date and Time: 2025-07-20T17:23:51.928661600

7. How to use map to convert objects into Uppercase in Java 8? Write the implementation and share the output screenshot.

We can use map() from the Stream API to transform each element of a collection. For example, converting a list of strings to uppercase:

Implementation:

```
package q7_case;
```

```
import java.util.Arrays;
```

```
import java.util.List;

import java.util.stream.Collectors;

public class UppercaseMap {

    public static void main(String[] args) {

        List<String> names = Arrays.asList("hema", "alex", "janani");

        List<String> upperNames = names.stream()

            .map(String::toUpperCase)

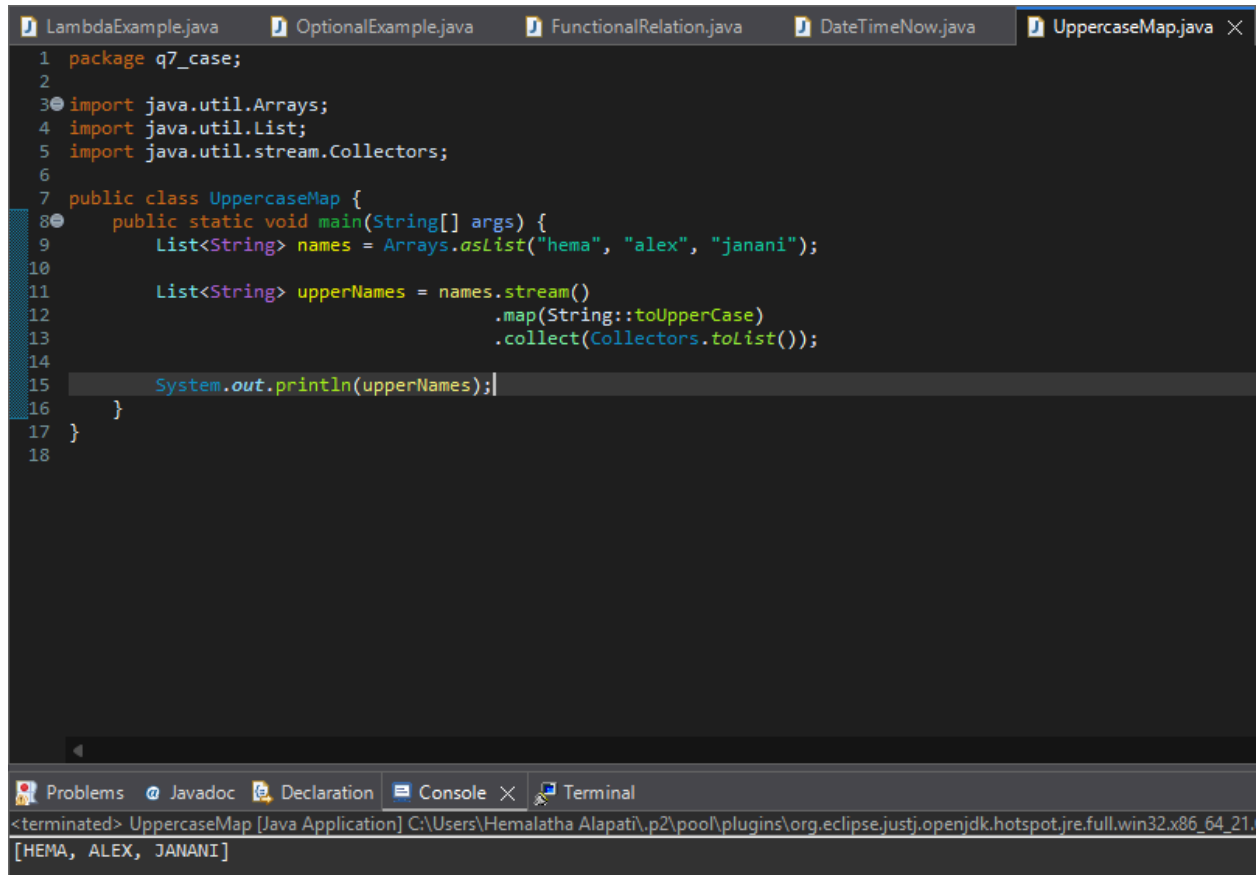
            .collect(Collectors.toList());

        System.out.println(upperNames);

    }

}
```

Output:



```
1 package q7_case;
2
3 import java.util.Arrays;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 public class UppercaseMap {
8     public static void main(String[] args) {
9         List<String> names = Arrays.asList("hema", "alex", "janani");
10
11         List<String> upperNames = names.stream()
12                                     .map(String::toUpperCase)
13                                     .collect(Collectors.toList());
14
15         System.out.println(upperNames);
16     }
17 }
18
```

The screenshot shows the Eclipse IDE with the 'UppercaseMap.java' file open. The code uses Java 8 streams to process a list of names. The console output at the bottom shows the result: [HEMA, ALEX, JANANI].

8. Explain how Java 8 has enhanced interface functionality with default and static methods. **Why were these features introduced, explain with a coding example?**

Before Java 8, interfaces could only have abstract methods (no implementation). Java 8 introduced two new features:

- **Default methods:** Allow interfaces to provide a default implementation, so classes don't always have to override them.
- **Static methods:** Allow utility or helper methods to be added directly in interfaces.

Why were they introduced?

- To add new methods to existing interfaces without breaking backward compatibility (especially useful in large libraries like List, Map, etc.).
- To support functional-style programming by enabling behavior sharing without inheritance.

Coding Example:

Interface:

```
package q8_interface;

interface MyInterface {

    default void greet() {

        System.out.println("Hello from default method");

    }

    static void info() {

        System.out.println("Static method inside interface");

    }

}
```

Implementation:

```
package q8_interface;

public class InterfaceTest implements MyInterface {

    public static void main(String[] args) {

        InterfaceTest obj = new InterfaceTest();

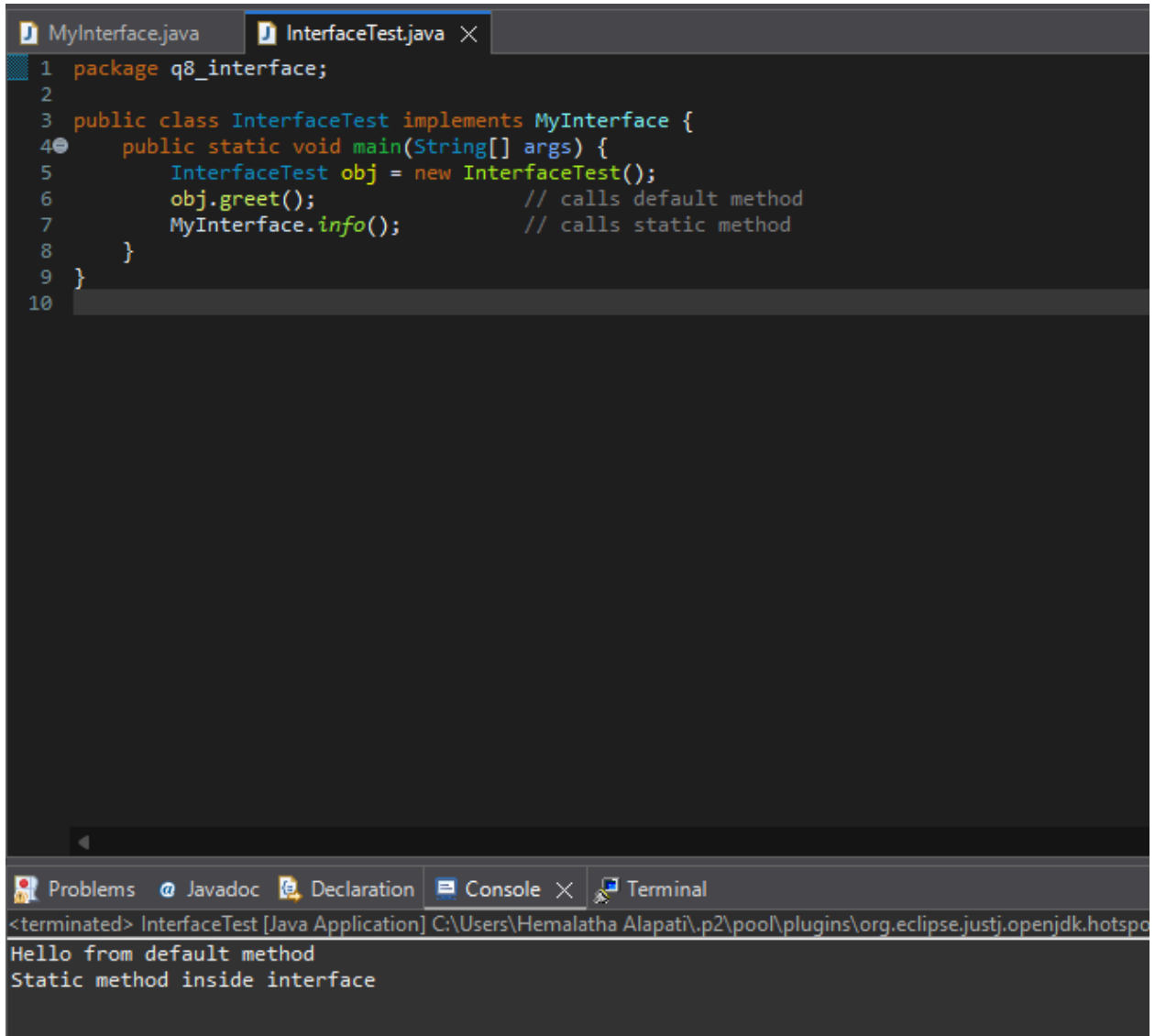
        obj.greet();           // calls default method

        MyInterface.info();    // calls static method

    }

}
```

Output:



```
1 package q8_interface;
2
3 public class InterfaceTest implements MyInterface {
4     public static void main(String[] args) {
5         InterfaceTest obj = new InterfaceTest();
6         obj.greet();           // calls default method
7         MyInterface.info();    // calls static method
8     }
9 }
10
```

Problems Javadoc Declaration Console × Terminal

<terminated> InterfaceTest [Java Application] C:\Users\Hemalatha Alapati\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot

Hello from default method
Static method inside interface

9. Discuss the significance of the Stream API introduced in Java 8 for data processing. How does it improve application performance and developer productivity?

The **Stream API** was one of the most powerful additions in Java 8. It allows for **declarative-style** processing of collections and supports operations like **filtering**, **mapping**, **sorting**, **reducing**, and more.

Why it's significant:

Simplifies code: You write less and cleaner code compared to traditional loops.

Boosts performance: Can easily switch to **parallel streams** for multicore processing.

Supports method chaining: Leads to more readable and fluent code.

Lazy evaluation: Intermediate operations are executed only when needed.

Example:

```
package q9_streamAPI;

import java.util.Arrays;

import java.util.List;

public class StreamExample {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

        long count = numbers.stream()

            .filter(n -> n % 2 == 0)

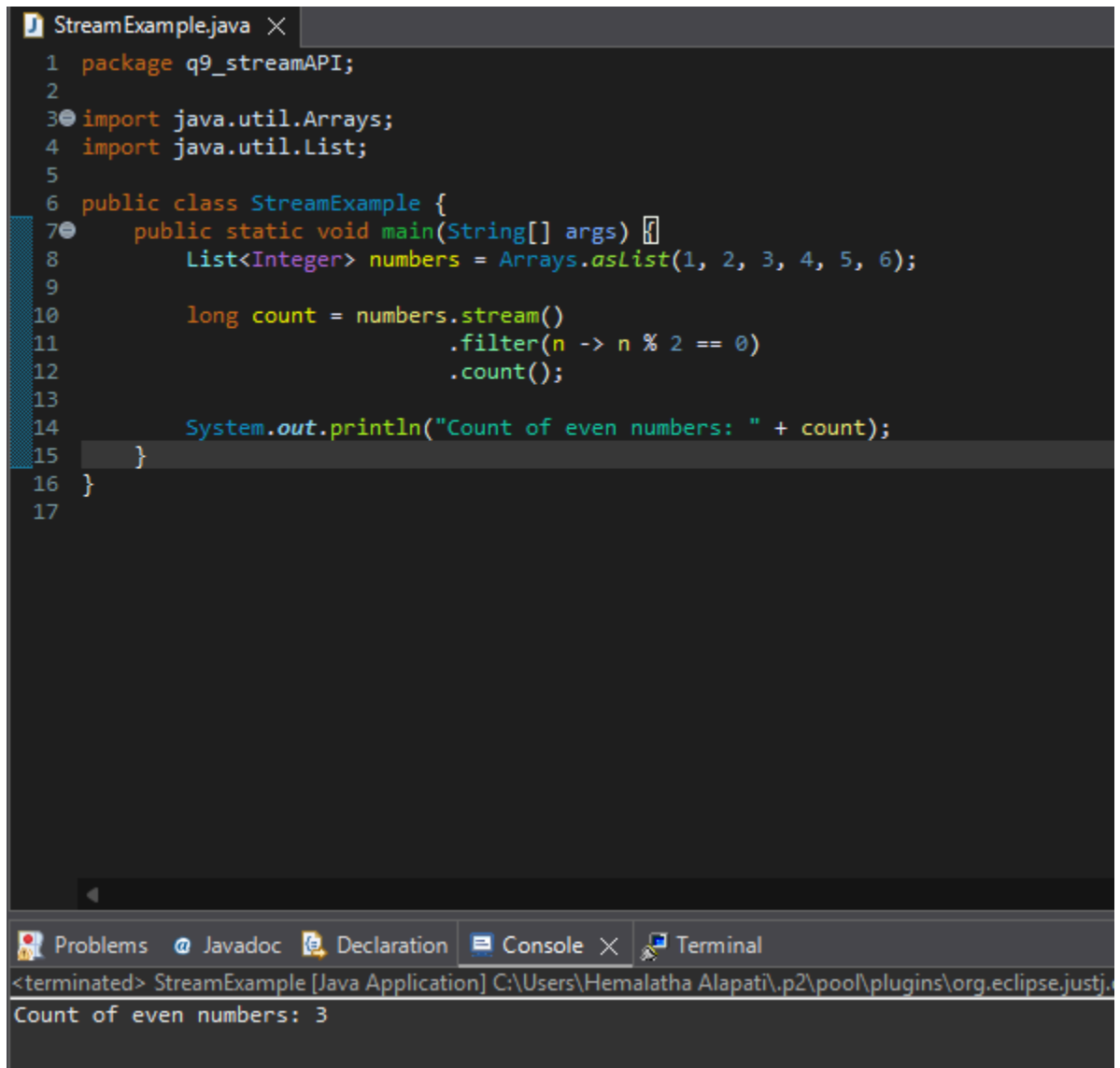
            .count();

        System.out.println("Count of even numbers: " + count);

    }

}
```

Output:



```
StreamExample.java X
1 package q9_streamAPI;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 public class StreamExample {
7     public static void main(String[] args) {
8         List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
9
10        long count = numbers.stream()
11                            .filter(n -> n % 2 == 0)
12                            .count();
13
14        System.out.println("Count of even numbers: " + count);
15    }
16 }
17
```

Problems Javadoc Declaration Console X Terminal

<terminated> StreamExample [Java Application] C:\Users\Hemalatha Alapati\.p2\pool\plugins\org.eclipse.justj.
Count of even numbers: 3

10. What are method references in Java 8, and how do they complement the use of lambda expressions? Provide an example where a method reference is more suitable than a lambda expression. **Explain with a coding example and share the output screenshot.**

Method references are a shorthand for lambda expressions that **call an existing method**. They make our code even more concise and readable.

How do they help?

When the lambda is just calling a method, you can directly use a method reference instead.

Coding Example where method reference is better:

```
package q10_methodReference;

import java.util.Arrays;

import java.util.List;

public class MethodRefExample {

    public static void printName(String name) {

        System.out.println(name);

    }

    public static void main(String[] args) {

        List<String> names = Arrays.asList("Hema", "Latha", "Java");

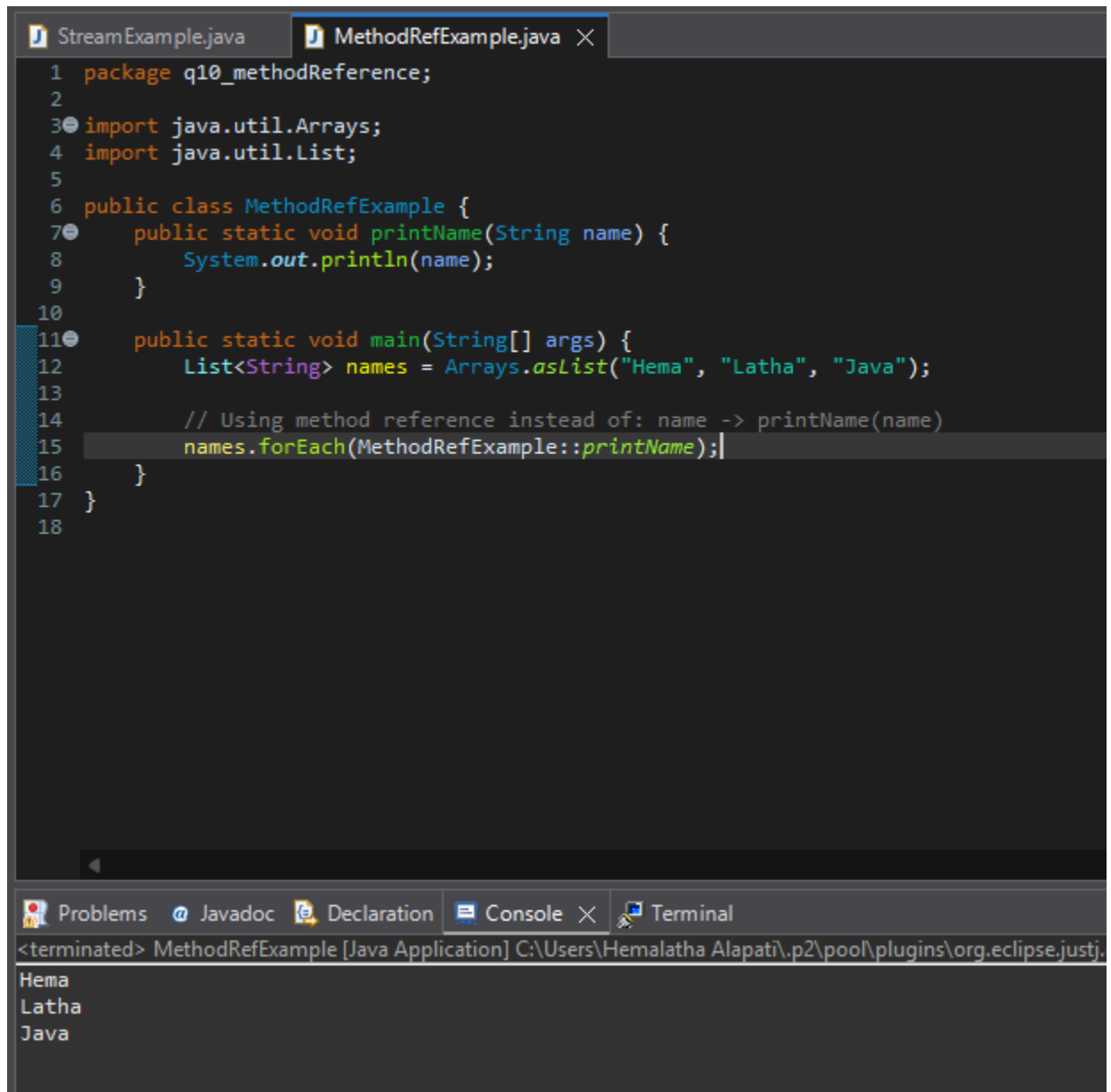
        // Using method reference instead of: name -> printName(name)

        names.forEach(MethodRefExample::printName);

    }

}
```


Output:



```
StreamExample.java MethodRefExample.java X
1 package q10_methodReference;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 public class MethodRefExample {
7     public static void printName(String name) {
8         System.out.println(name);
9     }
10
11     public static void main(String[] args) {
12         List<String> names = Arrays.asList("Hema", "Latha", "Java");
13
14         // Using method reference instead of: name -> printName(name)
15         names.forEach(MethodRefExample::printName);
16     }
17 }
18
```

Problems Javadoc Declaration Console X Terminal

<terminated> MethodRefExample [Java Application] C:\Users\Hemalatha Alapati\p2\pool\plugins\org.eclipse.justj.
Hema
Latha
Java