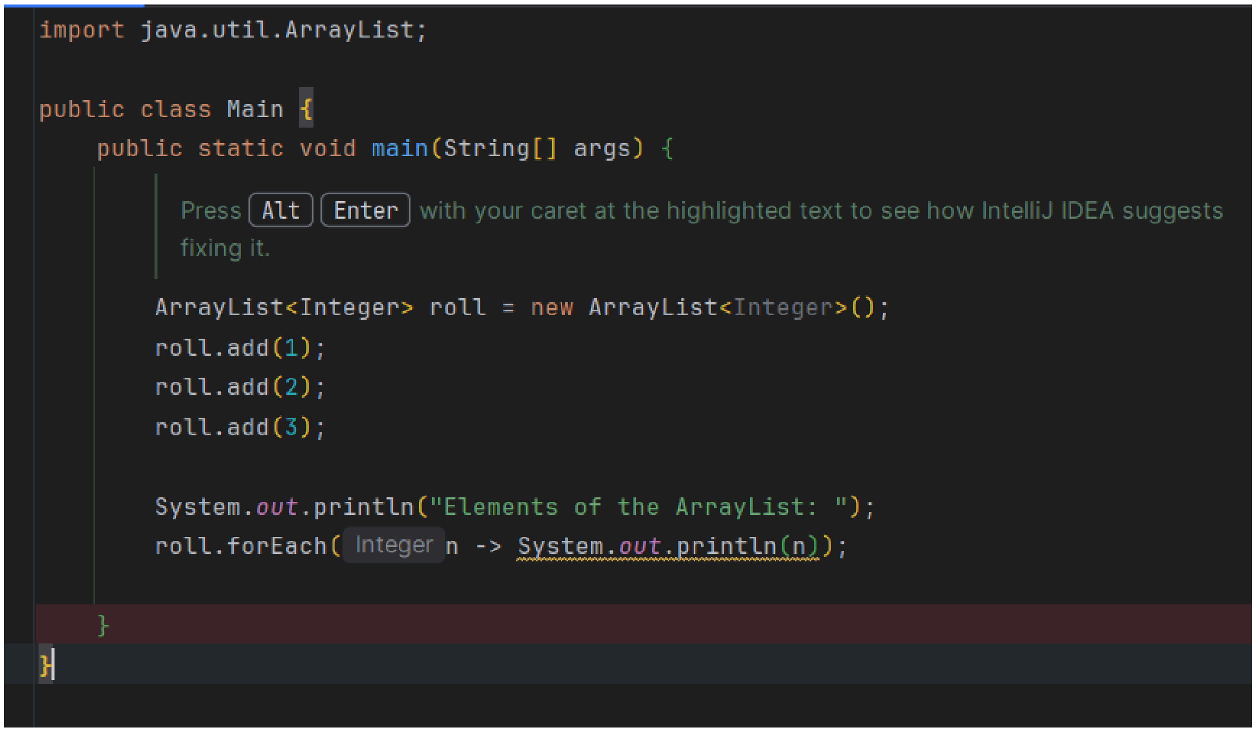
Ques 1) List the features of Java 8

- Lambda Expressions
- Functional Interfaces
- Stream API
- Optional Class
- Default and Static Methods in Interfaces

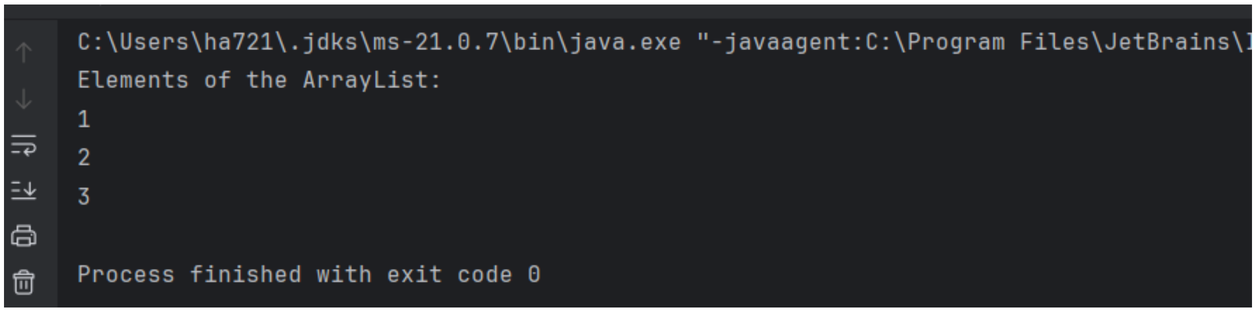
Ques 2) What is a Lambda Expression, and why do we use them? Explain with a coding example and share the output screenshot.

A Lambda expression in Java is a concise way to represent an anonymous function. It is a short block of code which takes in parameters and returns a value.

They are particularly useful for simplifying functional interfaces and enhancing code readability.



```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        Press Alt Enter with your caret at the highlighted text to see how IntelliJ IDEA suggests
        fixing it.
        ArrayList<Integer> roll = new ArrayList<Integer>();
        roll.add(1);
        roll.add(2);
        roll.add(3);

        System.out.println("Elements of the ArrayList: ");
        roll.forEach( Integer n -> System.out.println(n));

    }
}
```

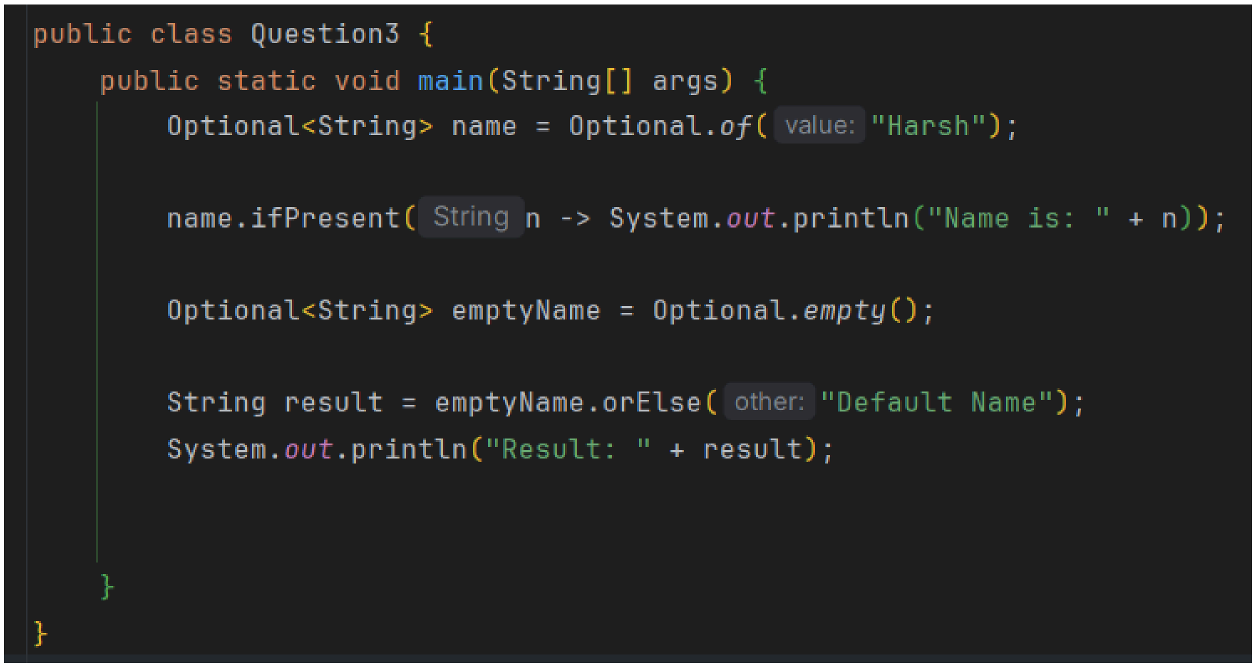


```
C:\Users\ha721\.jdks\ms-21.0.7\bin\java.exe "-javaagent:C:\Program Files\JetBrains\I
Elements of the ArrayList:
1
2
3

Process finished with exit code 0
```

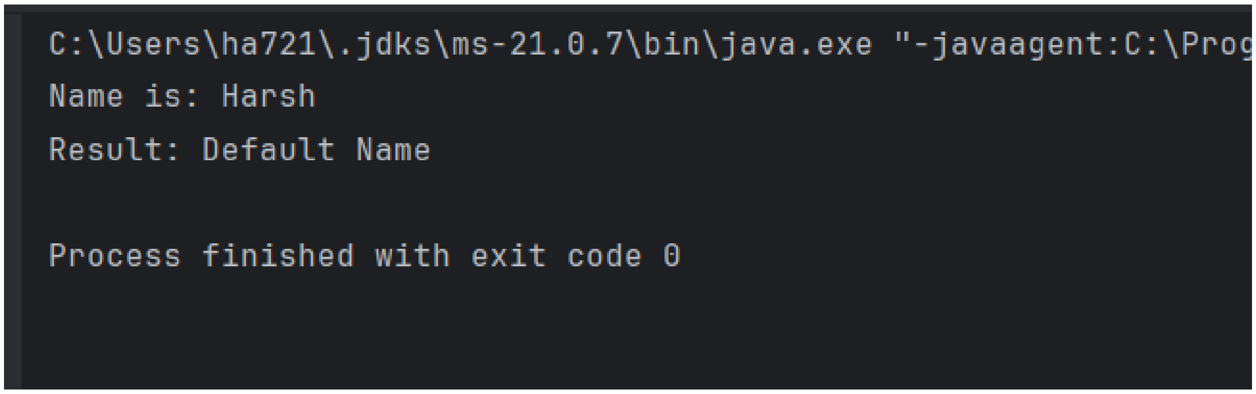
3. What is optional, and what is it best used for? Explain with a coding example and share the output screenshot.

Optional is a container class in Java 8 that may or may not hold a non-null value.

It helps you write null-safe code and avoid NullPointerException. It's designed to handle situations where a value might be null.



```
public class Question3 {
    public static void main(String[] args) {
        Optional<String> name = Optional.of( value: "Harsh");

        name.ifPresent( String n -> System.out.println("Name is: " + n));

        Optional<String> emptyName = Optional.empty();

        String result = emptyName.orElse( other: "Default Name");
        System.out.println("Result: " + result);


    }
}
```



```
C:\Users\ha721\.jdks\ms-21.0.7\bin\java.exe "-javaagent:C:\Prog
Name is: Harsh
Result: Default Name

Process finished with exit code 0
```

4. What is a Functional Interface? List some examples of predefined functional interfaces.
A functional interface is an interface that contains only one abstract method.
Some examples of predefined functional Interfaces are:-

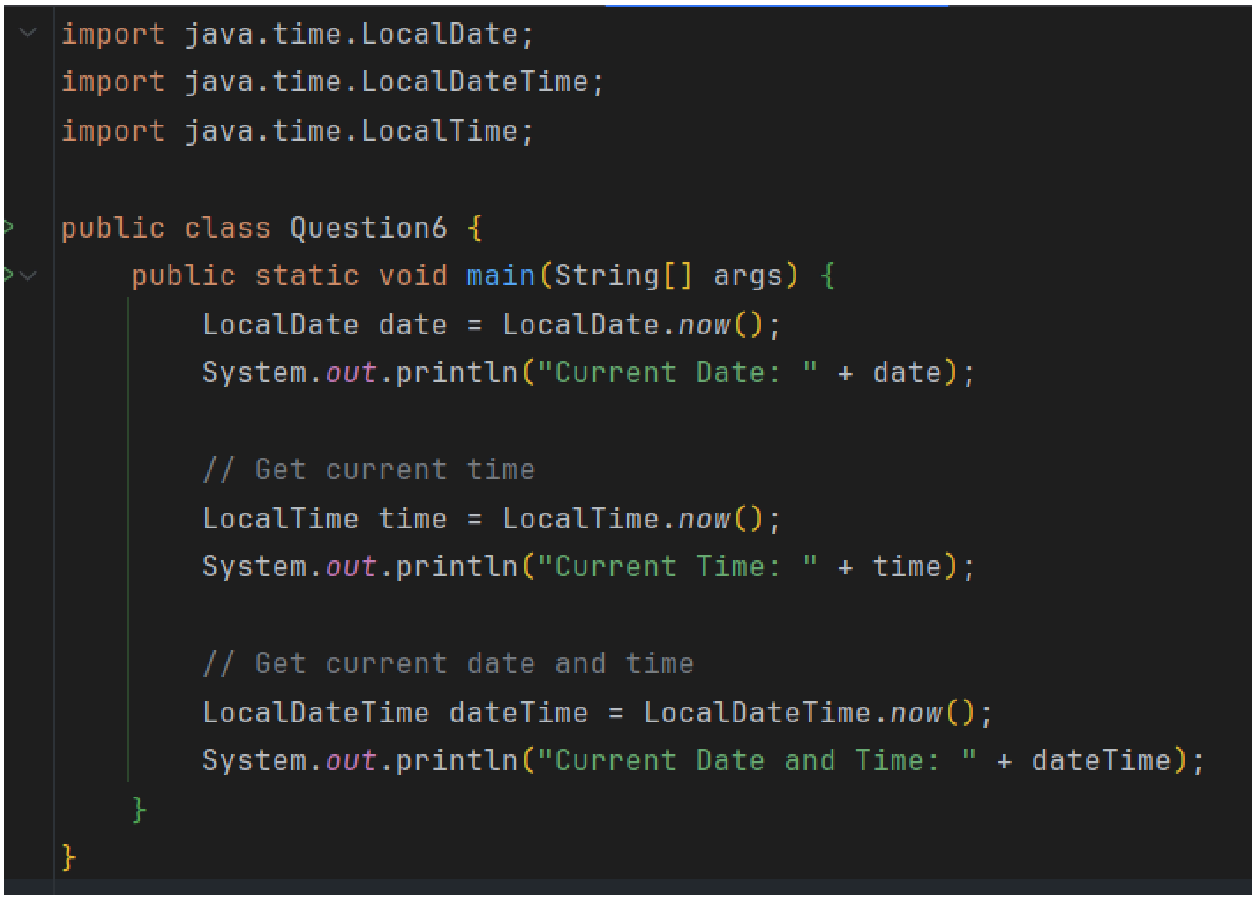
- Runnable – has method void run(): takes no arguments and returns nothing.
- Consumer<T> – has method void accept(T t): takes one argument and returns nothing.
- Supplier<T> – has method T get(): takes no argument and returns a result.
- Function<T, R> – has method R apply(T t): takes one argument and returns a result.

5. How are functional interfaces and Lambda Expressions related?
Functional interfaces and lambda expressions are closely related in Java because lambda expressions are used to provide concise implementations for functional interfaces. A functional interface is an interface with only one abstract method, and a lambda expression provides the implementation for that method.
This allows for more compact and readable code when working with single-method interfaces.

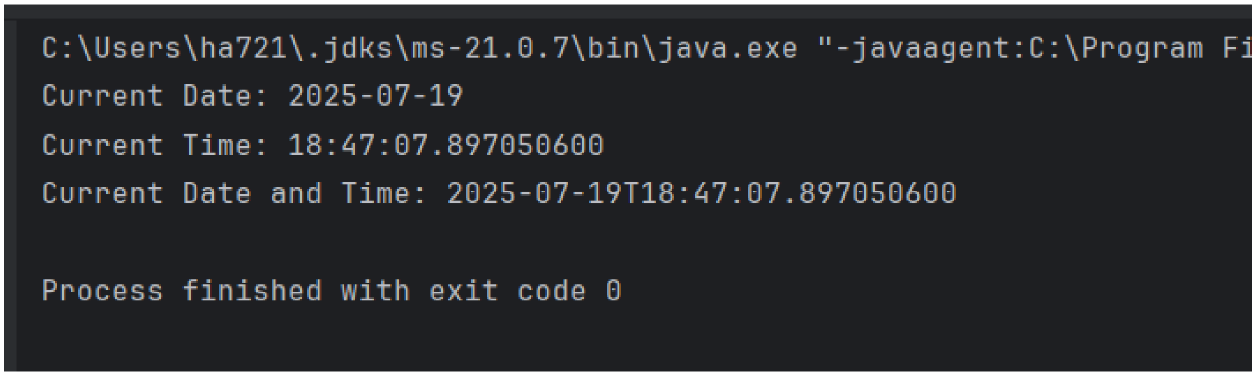
6. List some Java 8 Date and Time API's. How will you get the current date and time using Java 8 Date and Time API? Write the implementation and share the output screenshot.

Some Date and Time APIs are:-

- LocalDate – Represents only the date (e.g., 2025-07-19)
- LocalTime – Represents only the time (e.g., 13:45:20.123)
- LocalDateTime – Combines date and time (e.g., 2025-07-19T13:45:20.123)

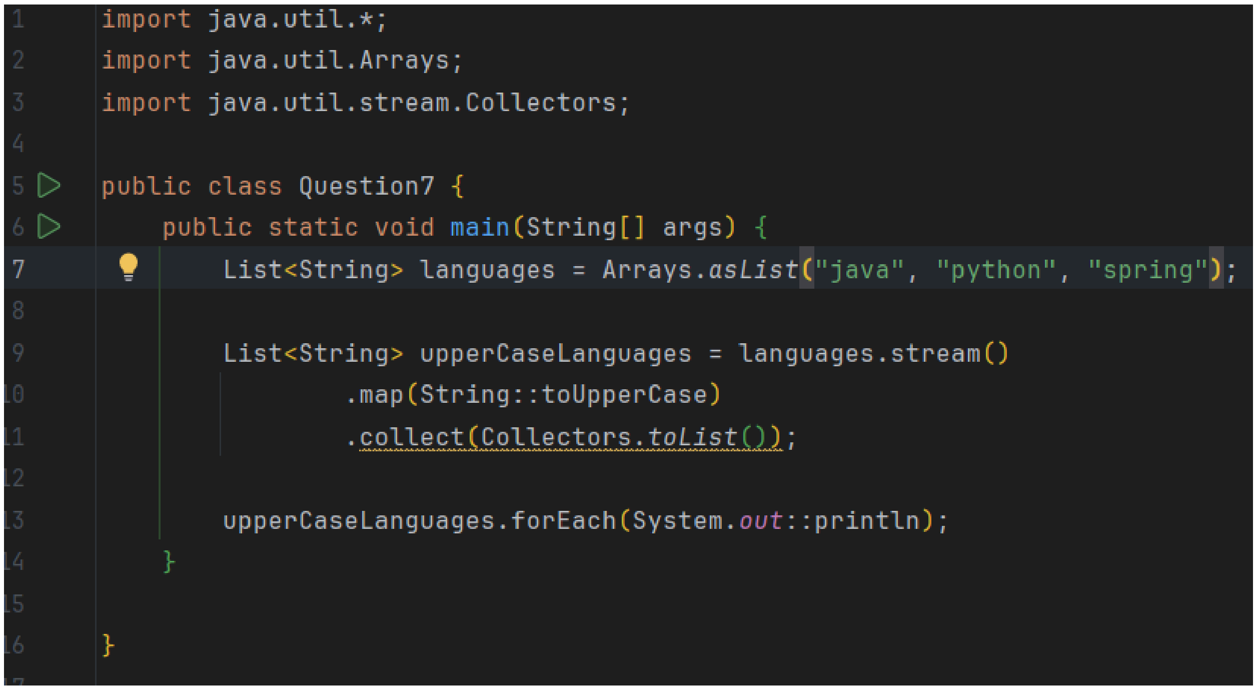


```java
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class Question6 {
    public static void main(String[] args) {
        LocalDate date = LocalDate.now();
        System.out.println("Current Date: " + date);

        // Get current time
        LocalTime time = LocalTime.now();
        System.out.println("Current Time: " + time);

        // Get current date and time
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("Current Date and Time: " + dateTime);
    }
}
```

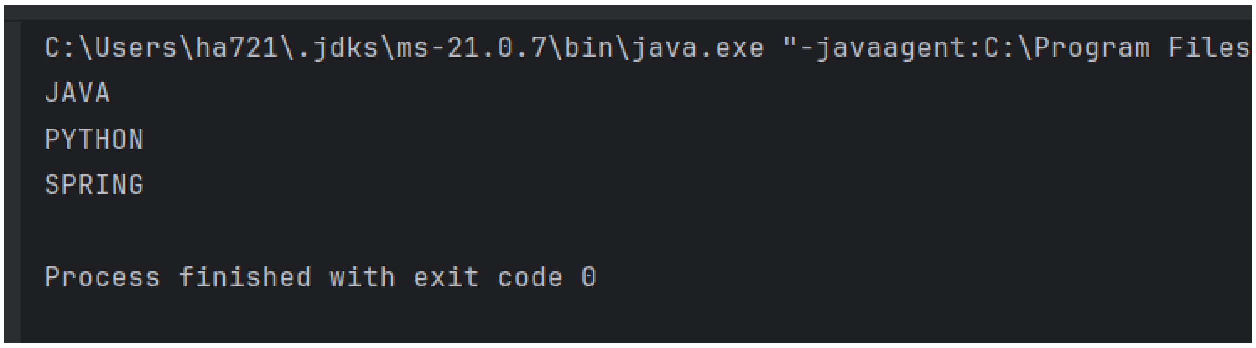


```
C:\Users\ha721\.jdks\ms-21.0.7\bin\java.exe "-javaagent:C:\Program Fi
Current Date: 2025-07-19
Current Time: 18:47:07.897050600
Current Date and Time: 2025-07-19T18:47:07.897050600

Process finished with exit code 0
```

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

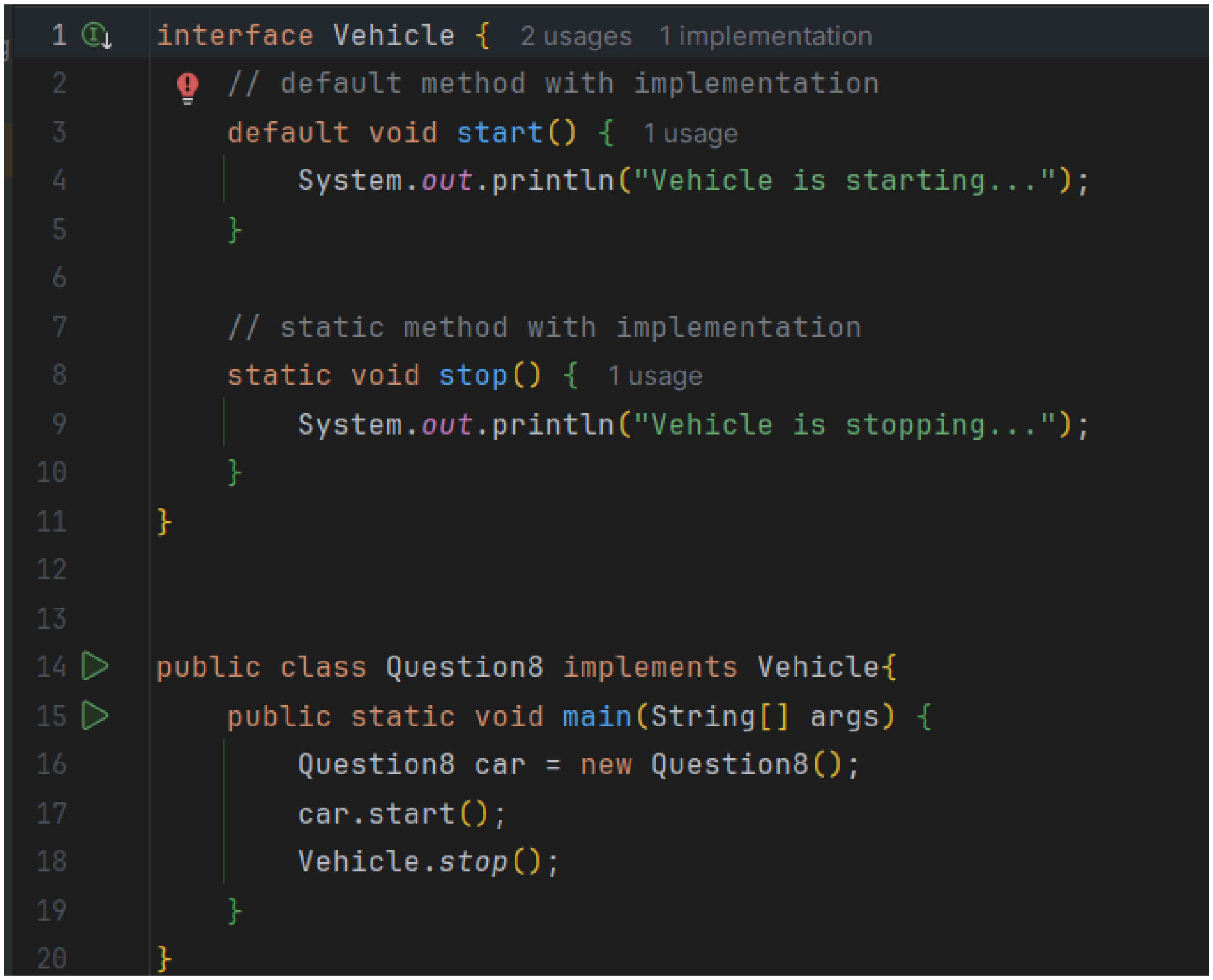


```java
import java.util.*;
import java.util.Arrays;
import java.util.stream.Collectors;

public class Question7 {
    public static void main(String[] args) {
        List<String> languages = Arrays.asList("java", "python", "spring");

        List<String> upperCaseLanguages = languages.stream()
                .map(String::toUpperCase)
                .collect(Collectors.toList());

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

}
```

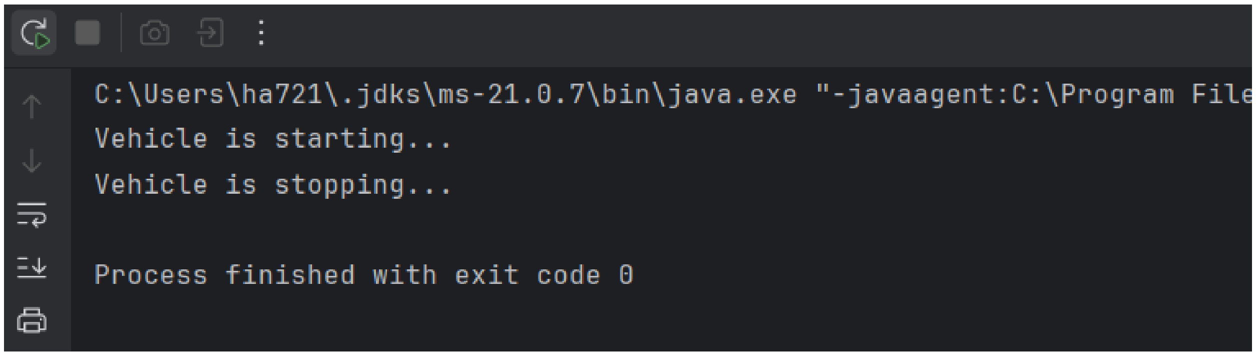


```
C:\Users\ha721\.jdks\ms-21.0.7\bin\java.exe "-javaagent:C:\Program Files
JAVA
PYTHON
SPRING

Process finished with exit code 0
```

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, making it hard to add new features without breaking existing code. Java 8 introduced default methods (with a body) to allow backward-compatible updates and static methods for utility operations within interfaces.



```java
interface Vehicle {  2 usages  1 implementation
    // default method with implementation
    default void start() {  1 usage
        System.out.println("Vehicle is starting...");
    }

    // static method with implementation
    static void stop() {  1 usage
        System.out.println("Vehicle is stopping...");
    }
}


public class Question8 implements Vehicle{
    public static void main(String[] args) {
        Question8 car = new Question8();
        car.start();
        Vehicle.stop();
    }
}
```



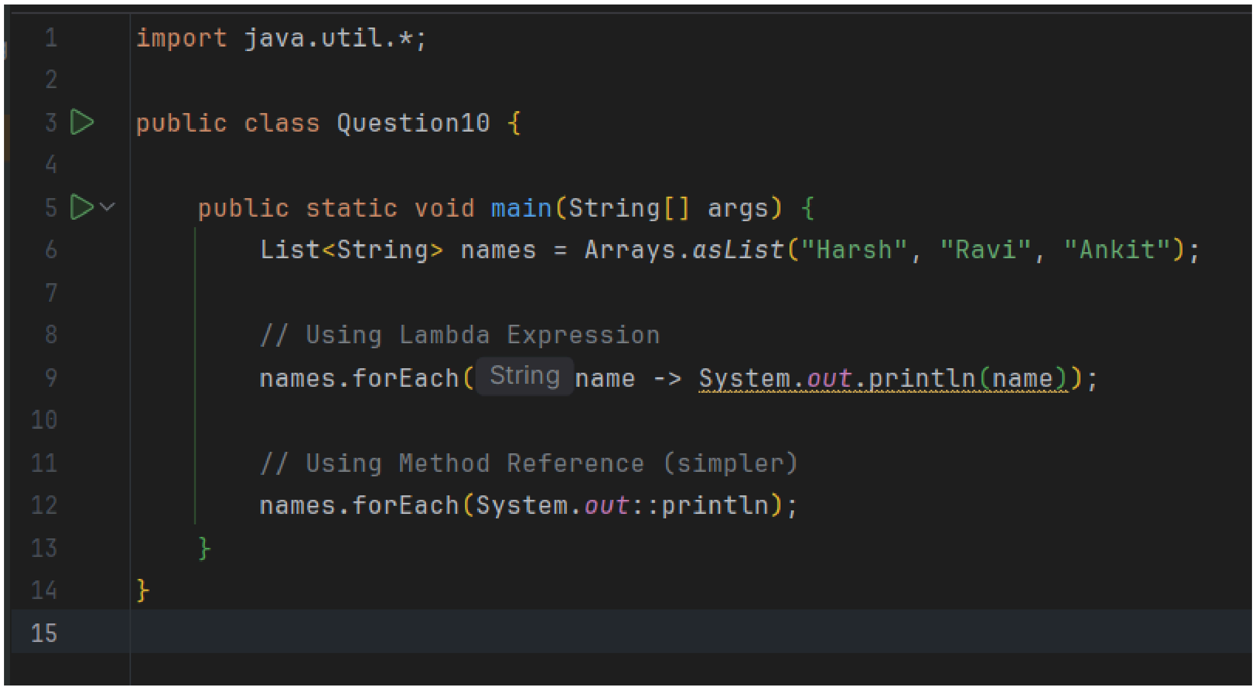
```
C:\Users\ha721\.jdks\ms-21.0.7\bin\java.exe "-javaagent:C:\Program File
Vehicle is starting...
Vehicle is stopping...

Process finished with exit code 0
```

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 in Java 8 allows efficient and declarative processing of collections using methods like map(), filter(), and collect(). It simplifies code, making it more readable and concise. Stream operations support lazy evaluation and can run in parallel, improving performance. It also promotes functional programming concepts like immutability and chaining of operations. Overall, it boosts both developer productivity and application efficiency.

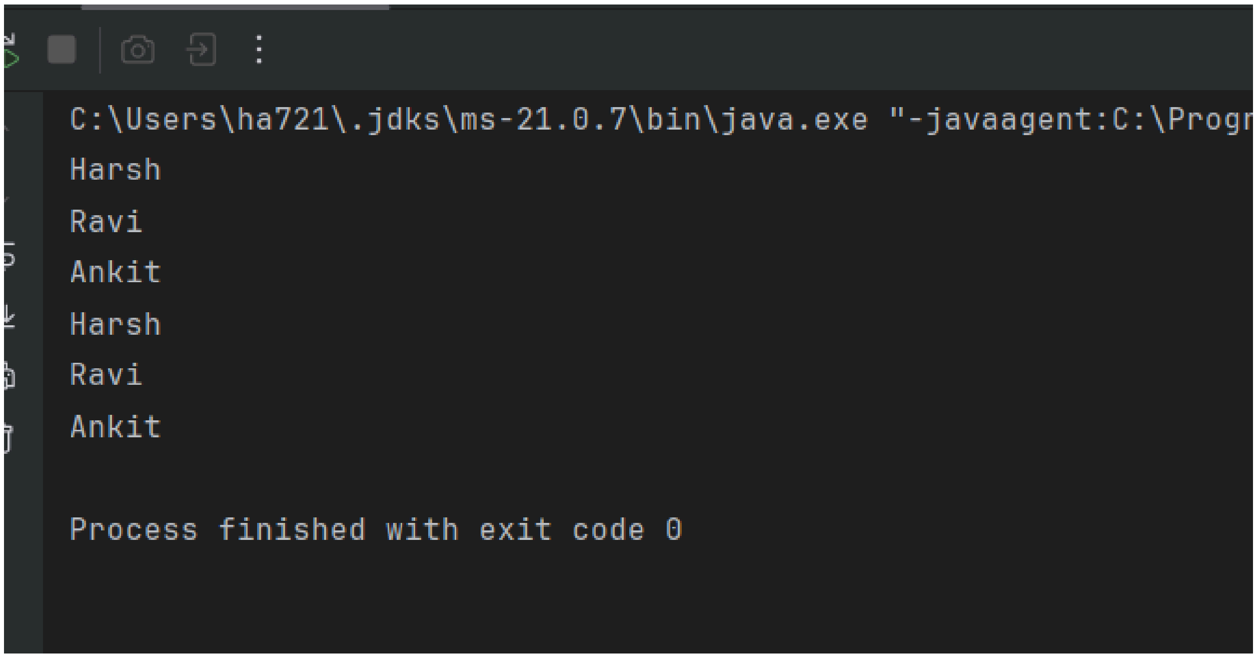
10. What are method references in Java 8, and how do they complement the use of lambda expressions? Provide an example where a method reference is more suitable than a lambda expression. Explain with a coding example and share the output screenshot.

Method References in Java 8 are a shorthand notation of lambda expressions to call methods directly using ::. They make code more concise and readable when a lambda just calls an existing method.

They complement lambda expressions by reducing boilerplate code and improving clarity when the lambda only invokes a method.



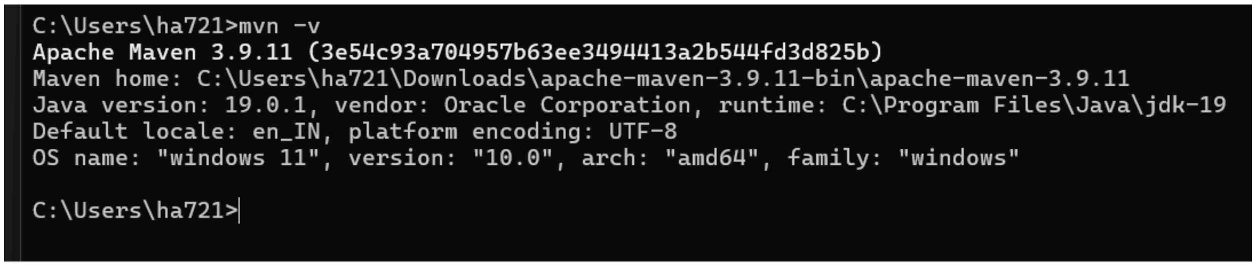
```java
import java.util.*;

public class Question10 {

    public static void main(String[] args) {
        List<String> names = Arrays.asList("Harsh", "Ravi", "Ankit");

        // Using Lambda Expression
        names.forEach( String name -> System.out.println(name));

        // Using Method Reference (simpler)
        names.forEach(System.out::println);
    }
}

```



```
C:\Users\ha721\.jdks\ms-21.0.7\bin\java.exe "-javaagent:C:\Progr
Harsh
Ravi
Ankit
Harsh
Ravi
Ankit

Process finished with exit code 0
```

Maven, Spring and Spring Boot

1. Install maven 3.6 or above. Execute mvn -v in the local terminal/command prompt and share the screenshot

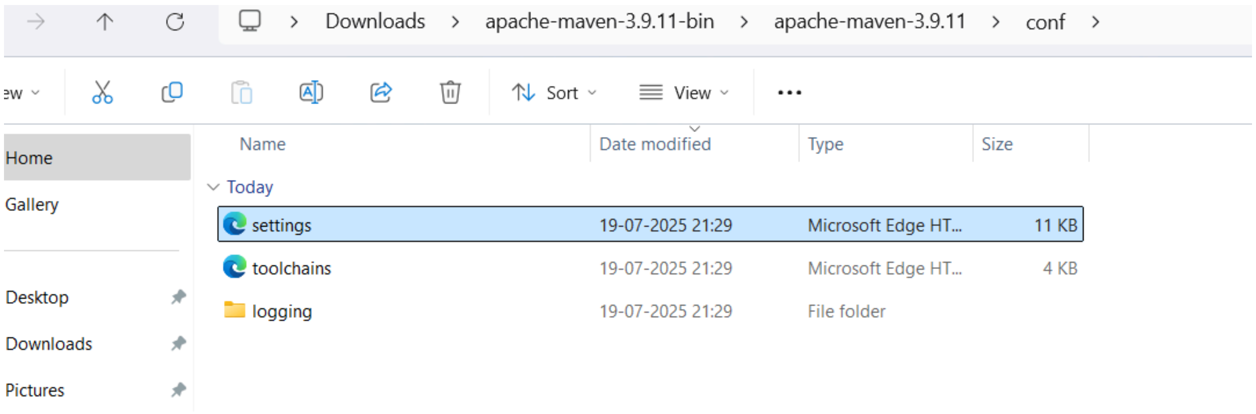


```
C:\Users\ha721>mvn -v
Apache Maven 3.9.11 (3e54c93a704957b63ee3494413a2b544fd3d825b)
Maven home: C:\Users\ha721\Downloads\apache-maven-3.9.11-bin\apache-maven-3.9.11
Java version: 19.0.1, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-19
Default locale: en_IN, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"

C:\Users\ha721>
```

2. What is the difference between maven central repository and local repository?
Maven Central Repository is a remote server that hosts publicly available libraries for Maven projects. Local Repository is a folder on your machine (~/.m2/repository) where Maven stores downloaded dependencies to avoid re-downloading them each time.

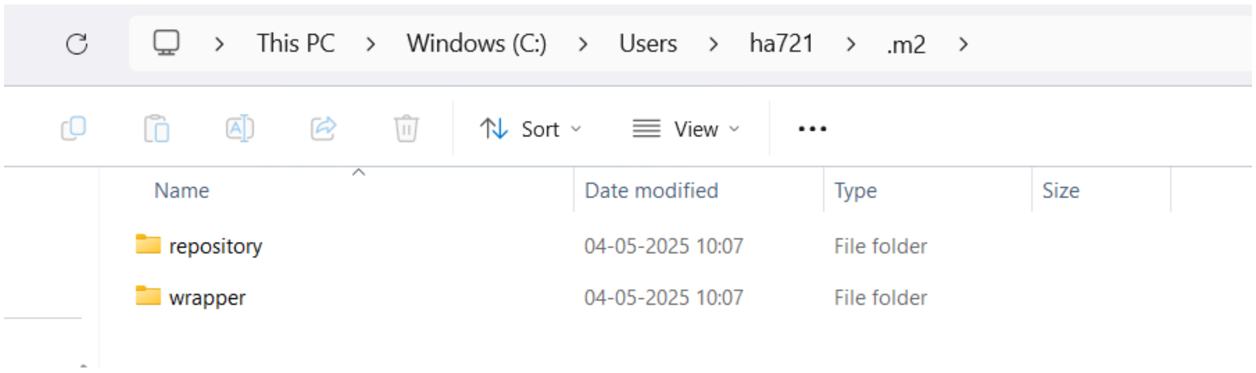
3. Maven commands
a. To build the maven project :- mvn clean install
b. To run the maven tests:- mvn test

4. Please locate the maven settings.xml file and local maven repository in your machine and share the screenshot



Downloads > apache-maven-3.9.11-bin > apache-maven-3.9.11 > conf >
Sort
View
Home
Gallery
Desktop
Downloads
Pictures

Name	Date modified	Type	Size
Today			
settings	19-07-2025 21:29	Microsoft Edge HT...	11 KB
toolchains	19-07-2025 21:29	Microsoft Edge HT...	4 KB
logging	19-07-2025 21:29	File folder	



Name	Date modified	Type	Size
repository	04-05-2025 10:07	File folder	
wrapper	04-05-2025 10:07	File folder	

5. The basic principle behind Dependency Injection(DI) is that the objects define their dependencies .What are the different ways in which an object can define its dependency ?

There are three main ways an object can define its dependencies in Dependency Injection (DI):
Constructor Injection – Dependencies are provided via the constructor.
Setter Injection – Dependencies are set through public setter methods.
Field Injection – Dependencies are injected directly into fields using annotations (e.g., @Autowired).

6. What is the difference between the @Autowired and @Inject annotation?

@Autowired is a Spring-specific annotation, while @Inject is part of Java's standard.
@Autowired can be used with required=false for optional injection.
@Inject does not support required=false directly.
Both work similarly for injecting dependencies, but @Autowired gives more Spring-specific features.

7. Explain the use of @Respository, @Component, @Service and @Controller annotations with an example for each.

@Component:- Generic annotation for any Spring-managed component.
Eg:-

```
import org.springframework.stereotype.Component;

@Component
public class MyHelper {
    public void assist() {
        System.out.println("Assisting...");
    }
}
```

@Repository:- This annotation is a specialization of @Component used to indicate that a class belongs to the persistence layer. It typically marks Data Access Objects (DAOs) or repository classes responsible for interacting with the database.

```
import org.springframework.stereotype.Repository;

@Repository
public class UserRepository {
    public void saveUser(String name) {
        System.out.println("User saved: " + name);
    }
}
```

@Service:- Used for service layer classes (business logic).

```
import org.springframework.stereotype.Service;

@Service
public class UserService {
    public String getUserInfo() {
        return "User Info";
    }
}
```

@Controller:- Used in Spring MVC to define web controllers. Handles HTTP requests and returns views.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class UserController {

    @GetMapping("/users")
    @ResponseBody
    public String getUsers() {
        return "List of users";
    }
}
```

8. Fix the code and explain why?

Correct Code:-

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class AppNamePrinter {

    @Value("${app.name}")  // ✅ Fix: use ${} to reference the property key
    private String appName;

    public void printAppName() {
        System.out.println("Application Name: " + appName);
    }
}
```

Reason:-

The original code uses @Value("app.name"), which injects the literal string "app.name" instead of fetching the value from application.properties. Using @Value("${app.name}") correctly references the property.

9. What does the @SpringBootApplication annotation do?
The @SpringBootApplication annotation is a combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan. It enables Spring Boot's auto-configuration and component scanning features in a single annotation.

10. What is the maven command to start the SpringBootApplication?
mvn spring-boot:run

11. Implement EmployeeCRUD using Spring and JDBC with the below Employee class. In the branch feature-spring, create a folder Employee-Spring. Push the solution to the branch and share the link.
https://github.com/harsharora1710/rg-assignments/tree/feature-spring

12. Implement EmployeeCRUD using SpringBoot and Spring Data JPA with the below Employee class. In the branch feature-spring, create a folder Employee-SpringBoot-JPA. Push the solution to the branch and share the link.

```
class Employee{
private int id;
private String name;
private String department;
}
```

https://github.com/harsharora1710/rg-assignments/tree/feature-spring

13. Follow the demo in the pre-work link https://www.youtube.com/watch?v=hr2XTbKSdAQ&t=18s and create a Spring Batch application that processes customer data. In the branch feature-spring, create a folder Customer-SpringBatch. Push the solution to the branch and share the link.
https://github.com/harsharora1710/rg-assignments/tree/main/Week%202/Assignment-01/Maven%20and%20Spring%20Boot/