

Assignment 2

Abhayjit Singh Gulati

1. List the features of Java 8

a. Lambda Expressions

- Enables treating functionality as a method argument or treating code as data.
- Simplifies the development of functional interfaces.

```
(List<String> list) -> list.size();
```

b. Functional Interfaces

- Interfaces with a single abstract method (SAM).
- Annotated with `@FunctionalInterface`.
- Example: `Runnable`, `Callable`, `Comparator`, `Function`, `Predicate`, etc.

c. Default Methods in Interfaces

- Allows interfaces to have method implementations.
- Useful for extending interfaces without breaking existing implementations.

```
default void log(String msg) {  
    System.out.println("Log: " + msg);  
}
```

d. Stream API

- Provides a high-level abstraction for processing collections in a functional style.
- Supports operations like `map`, `filter`, `reduce`, `collect`, etc.

```
List<String> filtered = list.stream()  
    .filter(s -> s.startsWith("A"))  
    .collect(Collectors.toList());
```

e. java.time API (Date and Time)

- A modern and comprehensive date and time API in java.time package.
- Replaces older classes like Date, Calendar, etc.

```
LocalDate today = LocalDate.now();  
LocalDate tomorrow = today.plusDays(1);
```

f. Optional Class

- Helps in handling null values gracefully and avoiding NullPointerException.

```
Optional<String> name = Optional.of("Java");  
  
name.ifPresent(System.out::println);
```

g. Method References

- A shorthand for calling methods using :: operator.
- Simplifies lambda expressions further.

```
list.forEach(System.out::println);
```

h. Collectors

- Used in combination with Streams for accumulating results.
- Example: Collectors.toList(), Collectors.joining(), Collectors.groupingBy(), etc.

i. Nashorn JavaScript Engine

- Allows execution of JavaScript code within Java applications.

```
ScriptEngine engine = new  
ScriptEngineManager().getEngineByName("nashorn");  
engine.eval("print('Hello from JavaScript');");
```

j. Parallel Streams

- Enables parallel processing of streams to leverage multi-core processors.
`list.parallelStream().forEach(System.out::println);`

k. **CompletableFuture and Enhanced Concurrency API**

- Asynchronous programming support through `CompletableFuture`.
- Provides better control over multithreading and non-blocking computations.
`CompletableFuture.supplyAsync(() -> "Hello")`
`.thenApply(s -> s + " World")`
`.thenAccept(System.out::println);`

l. **New Annotations**

- `@FunctionalInterface`
- `@Repeatable` (for repeating annotations)
- `@Target` enhancements to allow type annotations

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

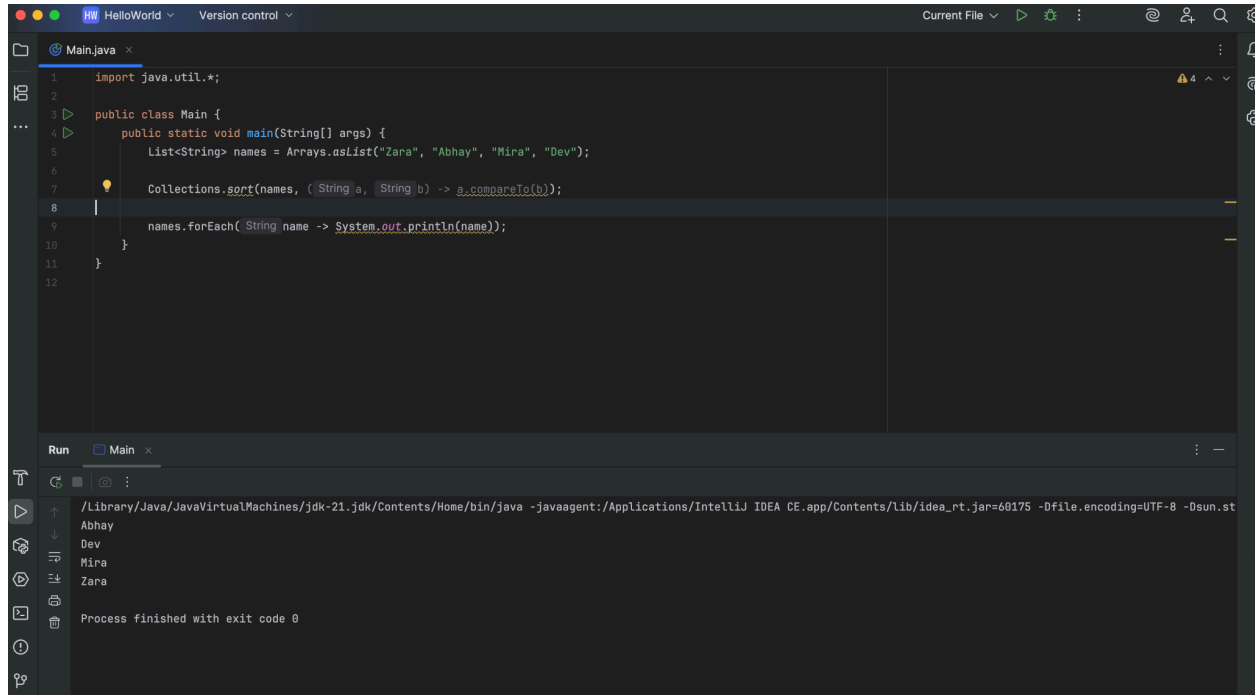
A **Lambda Expression** is a concise way to represent an **anonymous function**, a function that doesn't have a name and can be passed around as a value. Lambda expressions are primarily used to implement **functional interfaces** (interfaces with a single abstract method).

Lambda expressions in Java:

- Make code more readable and concise
- Enable functional programming in Java
- Allow passing behavior (methods) as arguments

- Replace anonymous inner classes in many scenarios, especially in collection processing or multithreading

Coding Example and Output



The screenshot displays the IntelliJ IDEA IDE. The top pane shows a Java file named `Main.java` with the following code:

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         List<String> names = Arrays.asList("Zara", "Abhay", "Mira", "Dev");
6
7         Collections.sort(names, (String a, String b) -> a.compareTo(b));
8
9         names.forEach( String name -> System.out.println(name));
10    }
11 }
12
```

The bottom pane shows the output of the program, which is the sorted list of names: Abhay, Dev, Mira, and Zara. The 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<T>` is a **container object** introduced in **Java 8** to represent a **potentially absent value**. It is used to avoid `NullPointerException` and provide a better alternative to using null references.

Optional helps with the following?

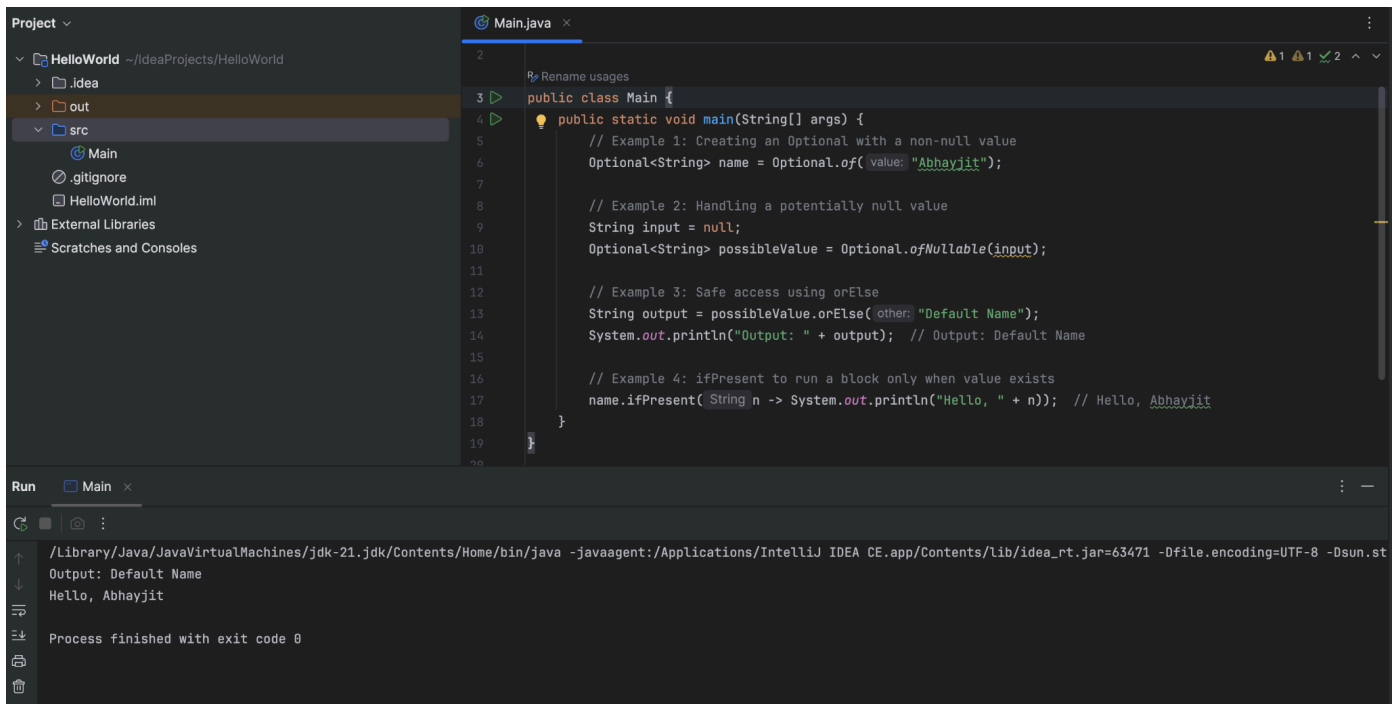
- Makes **null-checking** more explicit
- Promotes **functional-style programming**
- Helps avoid **runtime null pointer bugs**

- Improves code **readability and robustness**

Optional API Methods:

- Optional.of(value) – Creates an Optional with a non-null value
- Optional.ofNullable(value) – Allows null value
- Optional.empty() – Creates an empty Optional
- isPresent() – Checks if a value is present
- ifPresent(Consumer) – Executes action if value is present
- orElse(defaultValue) – Returns value if present, otherwise returns default
- map() – Transforms the value if present
- flatMap() – Like map, but avoids nested Optionals

Coding Example:



```
2
3 public class Main {
4     public static void main(String[] args) {
5         // Example 1: Creating an Optional with a non-null value
6         Optional<String> name = Optional.of( value: "Abhayjit");
7
8         // Example 2: Handling a potentially null value
9         String input = null;
10        Optional<String> possibleValue = Optional.ofNullable(input);
11
12        // Example 3: Safe access using orElse
13        String output = possibleValue.orElse( other: "Default Name");
14        System.out.println("Output: " + output); // Output: Default Name
15
16        // Example 4: ifPresent to run a block only when value exists
17        name.ifPresent( String n -> System.out.println("Hello, " + n)); // Hello, Abhayjit
18    }
19
20 }
```

Run Main x

```
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/Lib/idea_rt.jar=63471 -Dfile.encoding=UTF-8 -Dsun.st
Output: Default Name
Hello, Abhayjit
Process finished with exit code 0
```

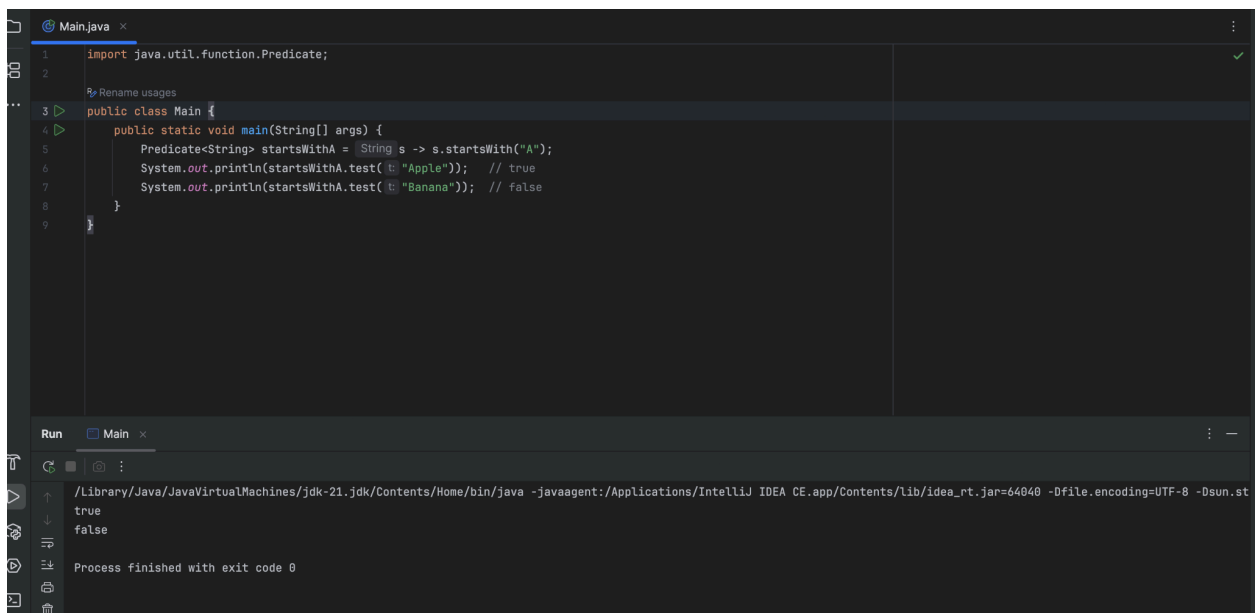
4. What is a functional interface? List some examples of predefined functional interfaces.

A functional interface is an interface with exactly one abstract method. It can have multiple default or static methods, but only one abstract method.

Functional interfaces are the foundation for lambda expressions and method references in Java.

Key Points:

- Annotated with `@FunctionalInterface` (optional but recommended)
- Used as targets for lambda expressions and method references
- Enforce a single method to implement functional programming style



```
1 import java.util.function.Predicate;
2
3 public class Main {
4     public static void main(String[] args) {
5         Predicate<String> startsWithA = s -> s.startsWith("A");
6         System.out.println(startsWithA.test("Apple")); // true
7         System.out.println(startsWithA.test("Banana")); // false
8     }
9 }

Run Main x
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=64040 -Dfile.encoding=UTF-8 -Dsun.st
true
false
Process finished with exit code 0
```

5. How are functional interfaces and Lambda Expressions related?

In Java, lambda expressions provide a clear and concise way to represent instances of functional interfaces. A functional interface is an interface that contains exactly one abstract method, and it serves as the target type for a lambda expression.

When a lambda expression is written, the Java compiler uses the context to infer the functional interface it is intended to implement. The lambda then provides the implementation of the abstract method defined in that interface.

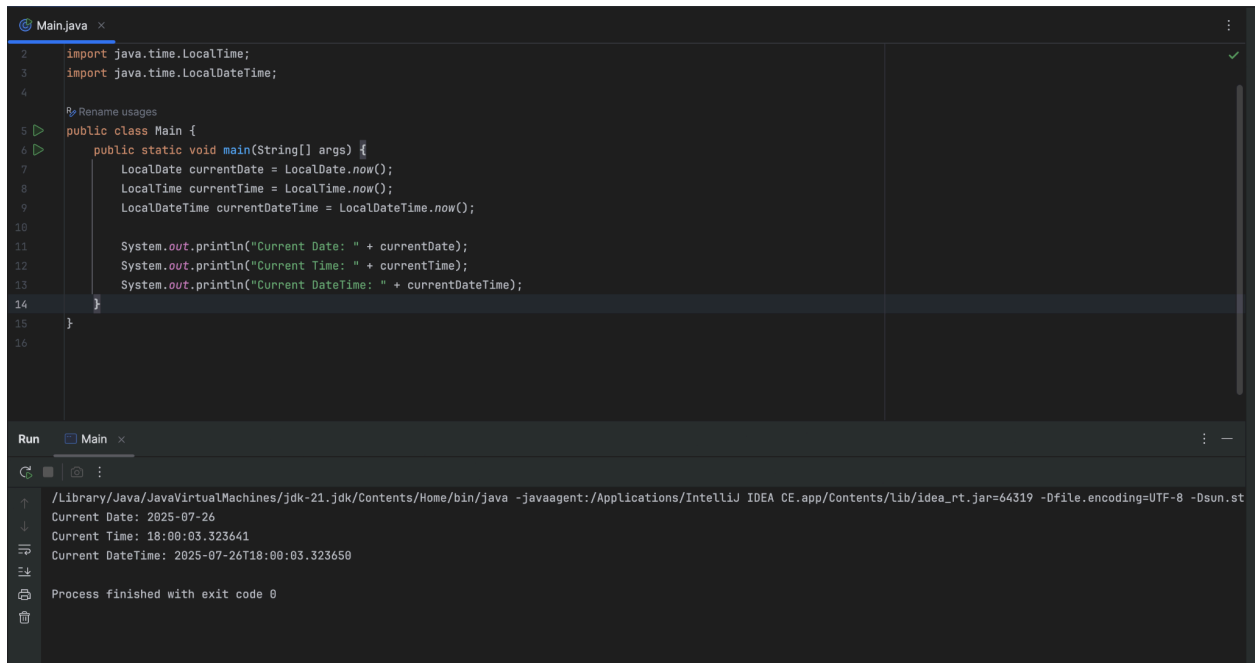
This relationship allows for writing cleaner, more expressive code, especially when working with APIs like the Java Stream API or custom callback functions.

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 and improved Date and Time API under the package `java.time`, which is more accurate, readable, and thread-safe compared to the older `Date` and `Calendar` classes.

Common Java 8 Date and Time APIs:

- `LocalDate` – Represents a date (year, month, day) without time.
- `LocalTime` – Represents a time (hour, minute, second, nanosecond) without a date.
- `LocalDateTime` – Represents a combination of date and time.
- `ZonedDateTime` – Date and time with a time zone.
- `Instant` – Represents a point in time (timestamp).
- `Period` – Represents a time-based amount of time in years, months, and days.
- `Duration` – Represents a time-based amount in seconds and nanoseconds.



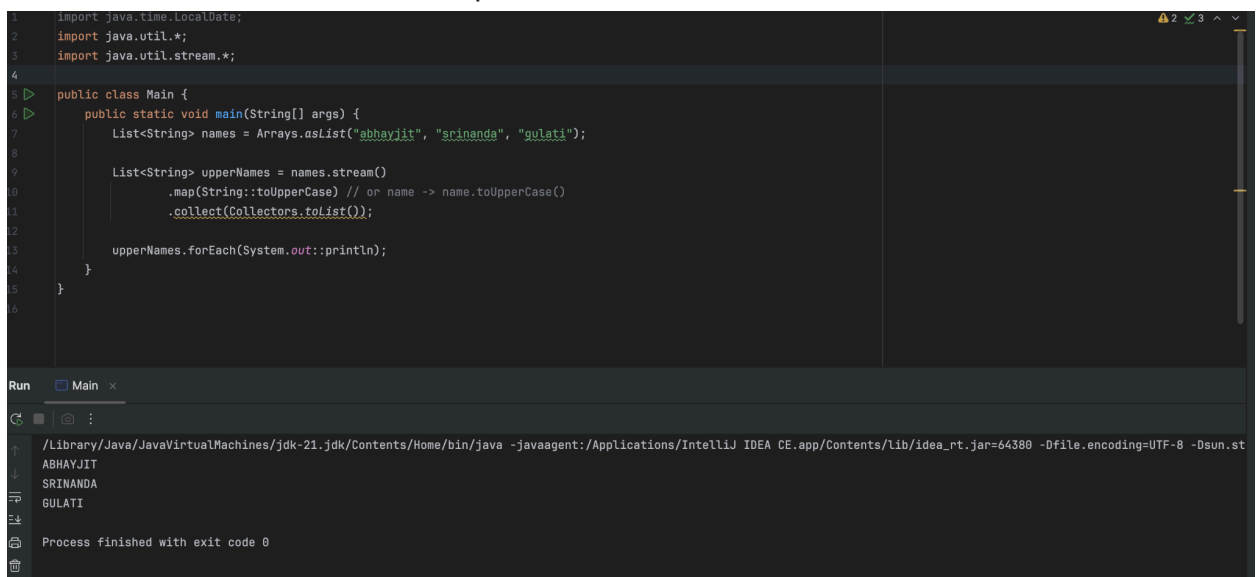
```
1 Main.java x
2 import java.time.LocalDateTime;
3 import java.time.LocalDate;
4
5 // Rename usages
6 public class Main {
7     public static void main(String[] args) {
8         LocalDate currentDate = LocalDate.now();
9         LocalTime currentTime = LocalTime.now();
10        LocalDateTime currentDateTime = LocalDateTime.now();
11
12        System.out.println("Current Date: " + currentDate);
13        System.out.println("Current Time: " + currentTime);
14        System.out.println("Current DateTime: " + currentDateTime);
15    }
16}
```

Run Main x

```
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=64319 -Dfile.encoding=UTF-8 -Dsun.st
Current Date: 2025-07-26
Current Time: 18:00:03.323641
Current DateTime: 2025-07-26T18:00:03.323650
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.

In Java 8, the `map()` method from the Stream API is used to transform elements of a collection. To convert strings to uppercase, we can use `map()` along with a method reference or a lambda expression.



```
1 import java.time.LocalDate;
2 import java.util.*;
3 import java.util.stream.*;
4
5 public class Main {
6     public static void main(String[] args) {
7         List<String> names = Arrays.asList("abhayjit", "srinanda", "gulati");
8
9         List<String> upperNames = names.stream()
10             .map(String::toUpperCase) // or name -> name.toUpperCase()
11             .collect(Collectors.toList());
12
13         upperNames.forEach(System.out::println);
14     }
15 }
```

Run Main x

```
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=64380 -Dfile.encoding=UTF-8 -Dsun.st
ABHAYJIT
SRINANDA
GULATI
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, explained with a coding example?

Java 8 Interface Enhancements

Java 8 introduced two major enhancements to interfaces:

1. Default Methods

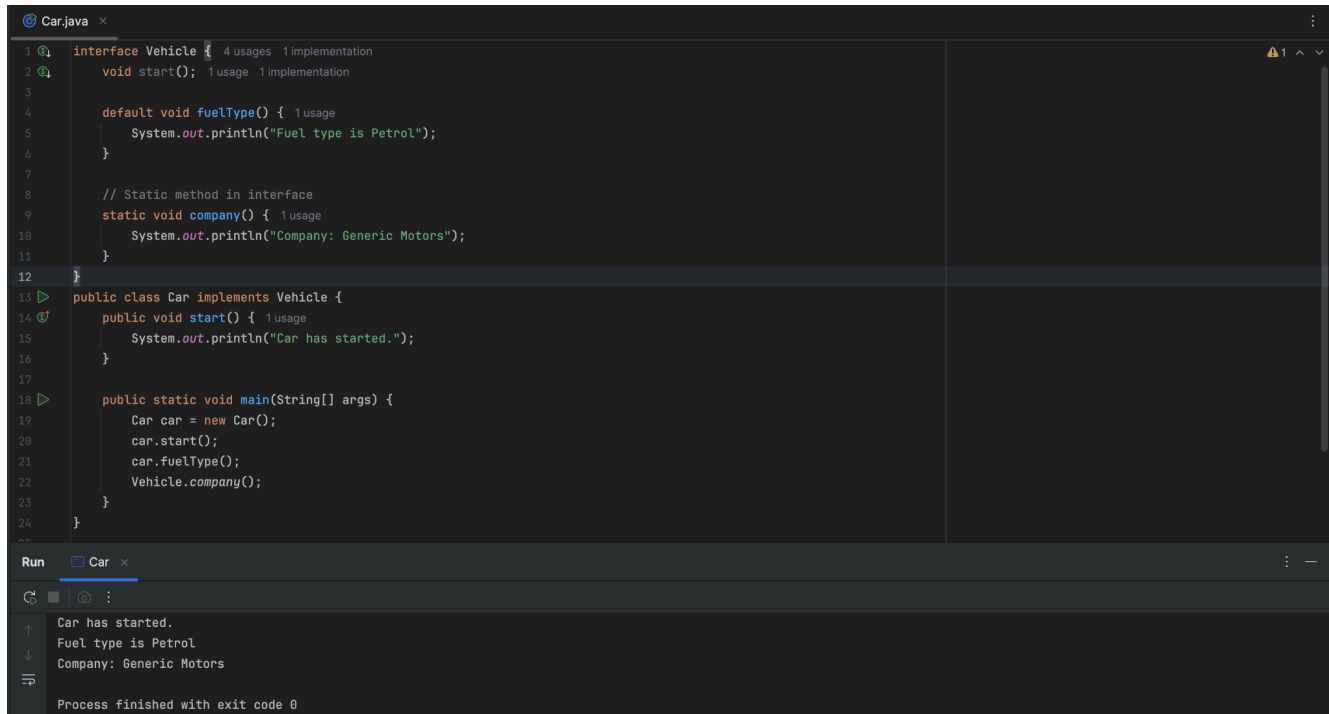
- Interfaces can now contain concrete methods with the default keyword.
- Allows providing a default implementation without affecting existing classes that implement the interface.

2. Static Methods

- Interfaces can also define static methods that belong to the interface itself (not instances).
- Can be called using `InterfaceName.method()`.

These Features were introduced:

- To support backward compatibility: Existing interfaces could evolve (e.g., new methods in `List` or `Collection`) without breaking the classes that implement them.
- To enable reusable logic in interfaces without forcing implementation in every class.
- To support functional programming with cleaner interfaces for lambda expressions and functional constructs.



```
1 interface Vehicle { 4 usages 1 implementation
2     void start(); 1 usage 1 implementation
3
4     default void fuelType() { 1 usage
5         System.out.println("Fuel type is Petrol");
6     }
7
8     // Static method in interface
9     static void company() { 1 usage
10         System.out.println("Company: Generic Motors");
11     }
12 }
13
14 public class Car implements Vehicle {
15     public void start() { 1 usage
16         System.out.println("Car has started.");
17     }
18
19     public static void main(String[] args) {
20         Car car = new Car();
21         car.start();
22         car.fuelType();
23         Vehicle.company();
24     }
25 }
```

Run Car x

```
Car has started.
Fuel type is Petrol
Company: Generic Motors

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?

Significance of Stream API in Java 8

The Stream API is one of the most powerful features introduced in Java 8 for processing collections of data in a declarative, functional-style.

Key Advantages

1. Simplifies Data Processing

- Allows developers to perform complex operations like filtering, mapping, sorting, and reducing in a concise and readable way.
- Encourages functional programming by chaining operations.

2. Improves Developer Productivity

- Reduces boilerplate code significantly.

- Enables writing clean, concise, and expressive code for bulk operations.

3. Supports Lazy Evaluation

- Intermediate operations are not executed until a terminal operation is invoked, improving resource usage.

4. Enables Parallel Processing

- With `.parallelStream()`, tasks can be processed in parallel across multiple cores, boosting application performance without much effort.



The screenshot shows an IDE window titled 'Main.java' with the following code:

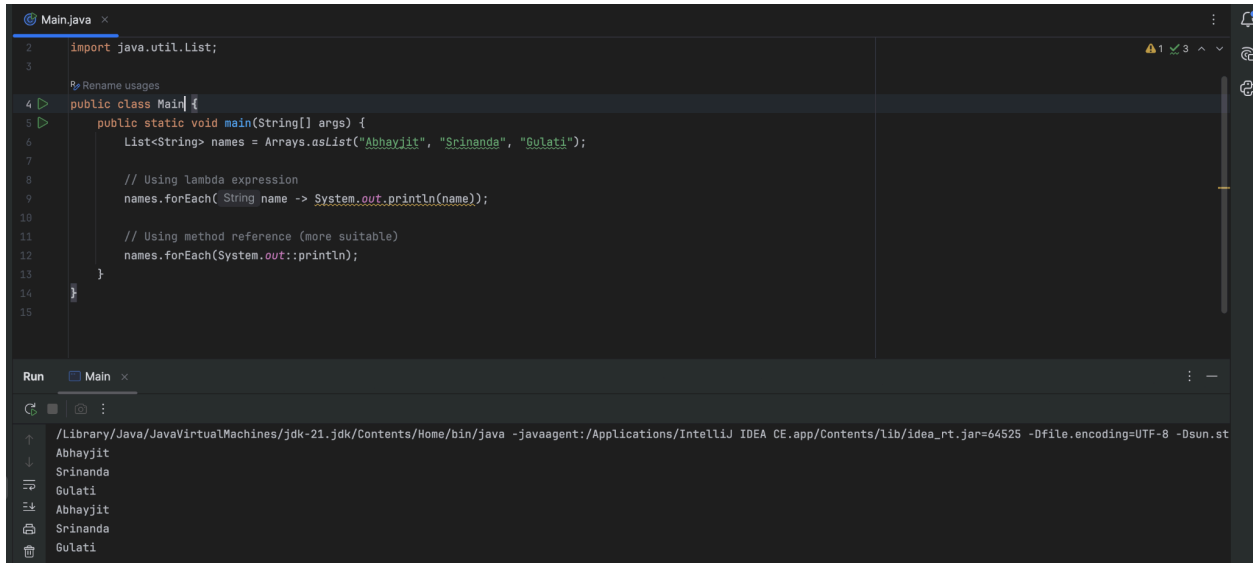
```
1 import java.util.Arrays;
2 import java.util.List;
3
4 public class Main {
5     public static void main(String[] args) {
6         List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
7
8         names.stream()
9             .filter( String name -> name.startsWith("A"))
10            .map(String::toUpperCase)
11            .forEach(System.out::println);
12     }
13 }
14
```

Below the code editor, the 'Run' tab is active, showing the command executed: `/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=64492 -Dfile.encoding=UTF-8 -Dsun.st`. The output of the program is displayed as 'ALICE'.

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.

In Java 8, method references provide a shorthand way to refer to methods using the `::` operator. They are a concise alternative to lambda expressions when the lambda simply calls an existing method.

Method references complement lambda expressions by making the code cleaner and more readable, especially when the lambda does nothing other than call a method.



```
1  Main.java x
2  import java.util.List;
3
4  public class Main {
5      public static void main(String[] args) {
6          List<String> names = Arrays.asList("Abhayjit", "Srinanda", "Gulati");
7
8          // Using lambda expression
9          names.forEach( String name -> System.out.println(name));
10
11         // Using method reference (more suitable)
12         names.forEach(System.out::println);
13     }
14 }
15
```

Run Main x

/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=64525 -Dfile.encoding=UTF-8 -Dsun.st

↑
↓
Abhayjit
Srinanda
Gulati
Abhayjit
Srinanda
Gulati

