1. Lambda Expressions

Lambda expressions allow you to write concise and functional-style code.

```
interface MathOperation {
    int add(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        MathOperation sum = (a, b) -> a + b;
        System.out.println(sum.add(3, 4));  // Output: 7
    }
}

// Lambda expression for Runnable interface.
Runnable runnable = () -> System.out.println("Hello, World from Lambda!");
runnable.run();
```

2. StringJoiner

StringJoiner is used to join strings with a delimiter.

```
import java.util.StringJoiner;

public class Main {
    public static void main(String[] args) {
        StringJoiner sj = new StringJoiner(", ");
        sj.add("Apple").add("Banana").add("Cherry");
        System.out.println(sj);  // Output: Apple, Banana, Cherry
    }
}
```

3. Functional Interface

A functional interface contains exactly one abstract method.

```
@FunctionalInterface
interface Greeting {
    void sayHello();  // Abstract method
}

public class Main {
    public static void main(String[] args) {
        Greeting greeting = () -> System.out.println("Hello, World!");  //
// out put - Lambda expression
        greeting.sayHello();
    }
}
```

Marker Interface

A marker interface has no methods, used to mark a class with some specific behavior. (Serialization , cloneble)

```
interface SerializableMarker {}

class Person implements SerializableMarker {
    private String name;
    public Person(String name){
       this.name = name;
        }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("John");
        if (person instanceof SerializableMarker) {
            System.out.println("Person is serializable.");
        }
    }
}
```

4. Stream API

Stream API allows functional-style operations on collections.

Filter method :

```
List<String> names = Arrays.asList("John", "Jane", "Jack", "sanket");
        names.stream()
             .filter(name -> name.startsWith("J"))
             .forEach(System.out::println);  // Output: John, Jane, 
Jack, Jill
```

Map method :

```
List<String> names = Arrays.asList("John", "Jane", "Jack", "sanket");

// Using map to convert all names to uppercase
        names.stream()
             .map(name -> name.toUpperCase())
             .forEach(System.out::println);
            // Output: JOHN, JANE, JACK, SANKET
```

5. Date and Time API

The new Date and Time API (`java.time`) provides more reliable date-time operations.

```
import java.time.LocalDate;

        LocalDate date = LocalDate.now();
        System.out.println(date);  // Output:Current Date (e.g.,2024-12-11)
    }
}
```

6. Static and Default Methods in Interfaces

Java 8 allows interfaces to have static and default methods.

```
interface MyInterface {
    default void defaultMethod() {
        System.out.println("Default Method");
    }

    static void staticMethod() {
        System.out.println("Static Method");
    }
}

public class Main {
    public static void main(String[] args) {
        MyInterface.staticMethod();  // Output: Static Method

        MyInterface obj = new MyInterface() {};
        obj.defaultMethod();  // Output: Default Method
    }
}
```

7. Optional Class

The **Optional** class was introduced in Java 8 to handle **null** values in a more structured way, avoiding the dreaded **`NullPointerException`.**

Example: Using `Optional` to avoid null checks

```
        String name = "Java";

        // Wrap a value in an Optional
        Optional<String> optionalName = Optional.ofNullable(name);

        // If the value is present, print it; else, print default message

        optionalName.ifPresentOrElse(
            value -> System.out.println("Hello, " + value),
            () -> System.out.println("Name is not available")
        );
    }
```

8. Method Reference

method reference is a shorthand notation of a lambda expression to call a method

```
import java.util.*;

public class MethodReferenceExample {

    // Static method
    public static void printMessage(String message) {
        System.out.println(message);
    }

    public static void main(String[] args) {
        List<String> messages = Arrays.asList("Hello", "World");

        // Using method reference to call static method

        messages.forEach(MethodReferenceExample::printMessage);
 // Prints each message
    }
}
```