Some common control constructs

Status	Done
Module	General
Reviewd	✓

1. `Optional` Class to Avoid Null Checks

The `Optional` class can be used to avoid null checks and NullPointerExceptions.

```
Optional<String> optional = Optional.ofNullable(possibleNullValue);
optional.ifPresent(value -> System.out.println("Value is present: " + value));
```

2. `Collectors` for Advanced Stream Operations

Use `Collectors` for advanced stream operations, such as grouping or partitioning data.

```
Map<Integer, List<String>> groupedByLength =
    strings.stream().collect(Collectors.groupingBy(String::length));
```

3. String Joining

Join strings with a delimiter easily using `String.join`.

```
List<String> list = Arrays.asList("A", "B", "C");
String result = String.join(",", list);
System.out.println(result);  // Output: A,B,C
```

4. Multi-Catch Exception Handling

Catch multiple exceptions in a single catch block.

```
try {
    // code that may throw IOException or SQLException
} catch (IOException | SQLException e) {
    e.printStackTrace();
}
```

5. Files.readAllLines and Files.write

Read and write files using `java.nio.file.Files`.

```
List<String> lines = Files.readAllLines(Paths.get("file.tx
t"));
Files.write(Paths.get("file.txt"), lines);
```

6. String Indentation

Easily add indentation to a string.

```
String indentedString = "Hello\nWorld".indent(4);
System.out.println(indentedString);
```

7. Var-args for Flexible Methods

Use var-args to allow methods to accept a variable number of arguments.

```
public void printAll(String... strings) {
    for (String s : strings) {
        System.out.println(s);
    }
}
```

8. Parallel Streams for Faster Processing

Use parallel streams for faster processing on large datasets.

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

9. Instant Class for Timestamps

Use the `Instant` class from `java.time` for timestamps.

```
Instant now = Instant.now();
System.out.println(now);
```

10. CompletableFuture for Asynchronous Programming

Use `CompletableFuture` for asynchronous programming.

```
CompletableFuture.supplyAsync(() -> "Hello")
                 .thenApply(result -> result + " World")
                 .thenAccept(System.out::println);
```

11. Pattern and Matcher for Advanced String Matching

Use `Pattern` and `Matcher` for advanced string matching.

```
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matcher("123abc456");
while (matcher.find()) {
    System.out.println(matcher.group());
}
```

12. Reflection for Dynamic Class Operations

Use reflection for dynamic operations on classes.

```
Class<?> clazz = Class.forName("com.example.MyClass");
Method method = clazz.getMethod("myMethod");
method.invoke(clazz.newInstance());

// clazz.newInstance() is a legacy method; in modern Java,
// clazz.getDeclaredConstructor().newInstance()   is preferre
d
```

When the above code runs:

1. `Class.forName("com.example.MyClass")` dynamically loads the `MyClass` class.
2. `clazz.getMethod("myMethod")` retrieves a reference to the `myMethod` method.
3. `method.invoke(clazz.newInstance())` creates an instance of `MyClass` and calls its `myMethod`

13. Method References for Clean Code

Use method references for cleaner and more readable code.

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

14. Try-With-Resources for Automatic Resource Management

Use try-with-resources to automatically close resources.

```
try (BufferedReader br = new BufferedReader(new FileReader("f
ile.txt"))) {
    br.lines().forEach(System.out::println);
}
```

15. Synchronized Collections

Use synchronized collections for thread-safe operations.

```
List<String> synchronizedList = Collections.synchronizedList
(new ArrayList<>());
synchronized (synchronizedList) {
    synchronizedList.add("Hello");
}
```

My Github Repo:

https://github.com/yousuf-git