Lab Exercise 1 - GitHub Copilot for Code Generation, Explanation, Refactoring, and Automated Documentation

1. Objective

This lab demonstrates how to use **GitHub Copilot** to:

- Generate code from natural language prompts
- Explain existing code
- Refactor and improve code quality
- Automatically generate documentation

The lab uses **Java (Spring Boot–style service)** as an example, but the steps apply to other languages like Python or JavaScript.

---

2. Prerequisites

- GitHub account with **GitHub Copilot enabled**
- IDE: **VS Code / IntelliJ IDEA** (with Copilot extension installed)
- Basic knowledge of Java

---

3. Lab Setup

Step 1: Enable GitHub Copilot

1. Open your IDE
2. Install **GitHub Copilot** extension
3. Sign in with your GitHub account
4. Verify Copilot is active (Copilot icon visible)

---

4. Part A: Code Generation Using Copilot

Task

Generate a Java service class that calculates the **total order amount** for a customer.

Step 1: Create a New Java File

Create a file named:

OrderService.java

Step 2: Write a Natural Language Prompt

Type the following comment and wait for Copilot suggestions:

```
// Create a service class to calculate total order amount for a customer

// Input: list of order amounts

// Output: total amount
```

Step 3: Accept Copilot Suggestion

Copilot will generate code similar to:

```
public class OrderService {
public double calculateTotal(List<Double> orders) {
double total = 0;
for (Double order : orders) {
total += order;
}
return total;
}
}
```

Outcome: Code is generated automatically from a comment.

---

5. Part B: Code Explanation Using Copilot

Task

Use Copilot to explain existing code.

Step 1: Select the Code

Highlight the calculateTotal method.

Step 2: Ask Copilot to Explain

Use Copilot Chat or comment:

```
// Explain what this method does
```

Example Explanation (Copilot Output)

- Iterates through a list of order values
- Adds each value to a running total
- Returns the final sum as the total order amount

Outcome: Copilot provides a clear, human-readable explanation.

---

6. Part C: Code Refactoring Using Copilot

Task

Refactor the code to use **Java Streams**.

Step 1: Add a Refactoring Prompt

Above the method, add:

```
// Refactor this method using Java Streams
```

Step 2: Accept Refactored Code

Copilot suggests:

```
public double calculateTotal(List<Double> orders) {
return orders.stream()
.mapToDouble(Double::doubleValue)
.sum();
}
```

Improvements:

- Cleaner code
- Better readability
- Functional programming style

---

7. Part D: Automated Documentation Generation

Task

Generate JavaDoc automatically using Copilot.

Step 1: Place Cursor Above Method

Type:

```
/**

*

*/
```

Step 2: Let Copilot Generate JavaDoc

Copilot generates:

```
/**
* Calculates the total amount of all orders for a customer.
*
* @param orders list of order amounts
* @return total order amount
*/
```

Outcome: Documentation created with minimal effort.

---

8. Part E: Copilot for Test Case Generation

Task

Generate a unit test for the service.

Step 1: Create Test File

OrderServiceTest.java

Step 2: Prompt Copilot

```
// Write a JUnit test for calculateTotal method
```

Example Output

```
@Test
void testCalculateTotal() {
OrderService service = new OrderService();
List<Double> orders = List.of(100.0, 200.0, 300.0);
assertEquals(600.0, service.calculateTotal(orders));
}
```

Outcome: Copilot generates a ready-to-use unit test.

---

9. Part F: Copilot for Code Improvement Suggestions

Task

Ask Copilot to improve robustness.

Prompt:

```
// Improve this code to handle null or empty lists
```

Copilot suggests:

```
public double calculateTotal(List<Double> orders) {
if (orders == null || orders.isEmpty()) {
return 0.0;
}
return orders.stream()
.mapToDouble(Double::doubleValue)
.sum();
}
```

Outcome: Safer, production-ready code.

---

10. Observations and Learning Outcomes

- GitHub Copilot accelerates **code writing**

- Helps **understand unfamiliar code**
- Encourages **best practices and refactoring**
- Reduces effort in **documentation and testing**
- Acts as an AI pair programmer, not a replacement

---

11. Conclusion

This lab demonstrated how GitHub Copilot can be effectively used for **code generation, explanation, refactoring, and automated documentation**. By leveraging AI-assisted development, developers can improve productivity, code quality, and maintainability while focusing on design and problem-solving.