Test-Driven Development (TDD) and its importance:

Test-Driven Development (TDD) is a software development approach where you write tests for your code before you actually implement the code itself. The TDD process typically follows these steps:

1. **Write a Test**: You start by writing a test case that defines the expected behavior of the code you are going to write. This test is supposed to fail initially because the code it is testing doesn't exist yet.

2. **Write the Code**: You then implement the code to make the test pass. Your goal is to make the test case pass with the minimum amount of code necessary.

3. **Refactor (if necessary)**: After the test passes, you can refactor the code to improve its structure or performance while ensuring the test still passes.

4. **Repeat**: You repeat this process for every piece of functionality you want to add or modify in your software.

The primary benefits of TDD include:

1. **Improved Code Quality**: TDD forces you to think about the desired behavior of your code before you write it. This leads to code that is more robust, reliable, and easier to maintain.

2. **Faster Debugging**: When a test fails, you know exactly where the problem is, which makes debugging faster and more straightforward.

3. **Regression Testing**: By running your tests frequently, you can catch regressions (new bugs introduced while modifying existing code) early in the development process.

4. **Documentation**: Tests serve as living documentation of your code, helping other developers understand its intended behavior.

5. **Confidence**: TDD gives you confidence that your code works as expected. If all tests pass, you can be reasonably sure that the code is functioning correctly.

Let's illustrate TDD with two Java code examples:

Example 1: Implementing a Simple Calculator

Suppose you want to create a basic calculator that can add two numbers. You start with a test case:

```java
```

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {
    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(3, 4);
        assertEquals(7, result);
    }
}
```

In this example, you haven't implemented the `Calculator` class or the `add` method yet. When you run this test, it will fail because the `Calculator` class and the `add` method don't exist. Now you can implement the code to make the test pass:

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

After implementing the code, you run the test again, and it should pass. If it passes, you have successfully implemented the addition functionality. You can now add more tests and functionality for subtraction, multiplication, etc.

Example 2: Implementing a Stack

Let's say you want to implement a stack data structure using TDD. You start with a test case:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class StackTest {
    @Test
    public void testPushAndPop() {
        Stack<Integer> stack = new Stack<>();
        stack.push(5);
        stack.push(10);
        assertEquals(10, stack.pop());
        assertEquals(5, stack.pop());
    }
}
```

This test will fail initially because you haven't implemented the `Stack` class yet. Now you can implement the code to make the test pass:

```java
import java.util.ArrayList;
import java.util.List;

public class Stack<T> {
    private List<T> elements = new ArrayList<>();

    public void push(T item) {
        elements.add(item);
    }

    public T pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty.");
        }
        return elements.remove(elements.size() - 1);
    }

    public boolean isEmpty() {
        return elements.isEmpty();
    }
}
```

After implementing the `Stack` class, you run the test again, and it should pass. You can then add more test cases and functionality for your stack, ensuring it behaves as expected.

In both examples, TDD helped ensure that the code met the specified requirements and was thoroughly tested before being used in a larger application. This approach leads to more reliable and maintainable code in the long run.