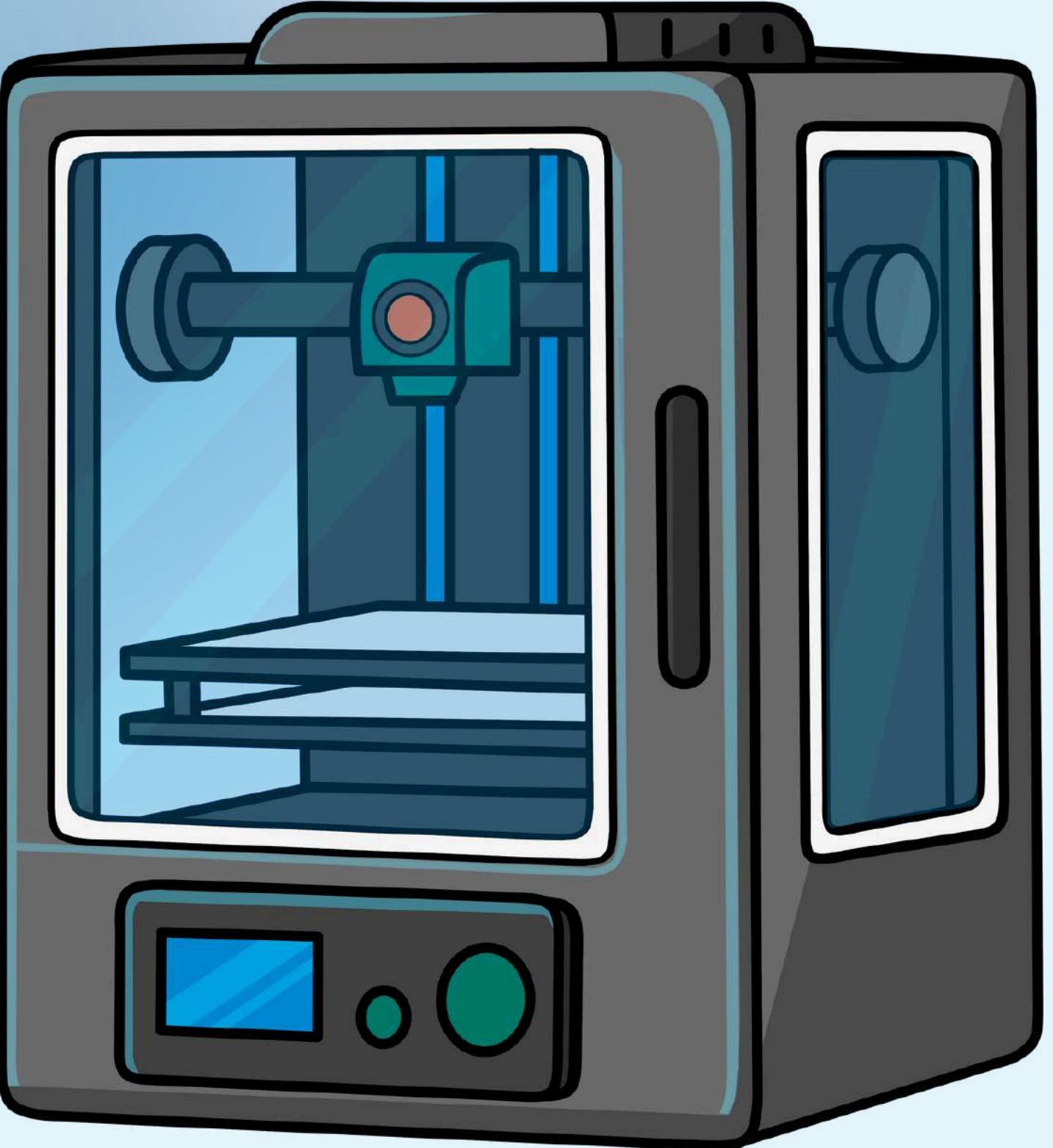


JUnit & Unit Testing

Nazanin Foroutan

AP - Fall 2025 - Dr. Vahidi Asl





Every Product Needs Testing

- 🌐 **No product is perfect from the start**
- 🌐 **A bug found late = very high cost**
- 🌐 **Famous failures:**
 - Samsung Note 7 – battery explosions
 - Ford Pinto – fuel tank fire hazard
- 🌐 **Importance of early defect detection**

Product Lifecycle & the Role of Testing



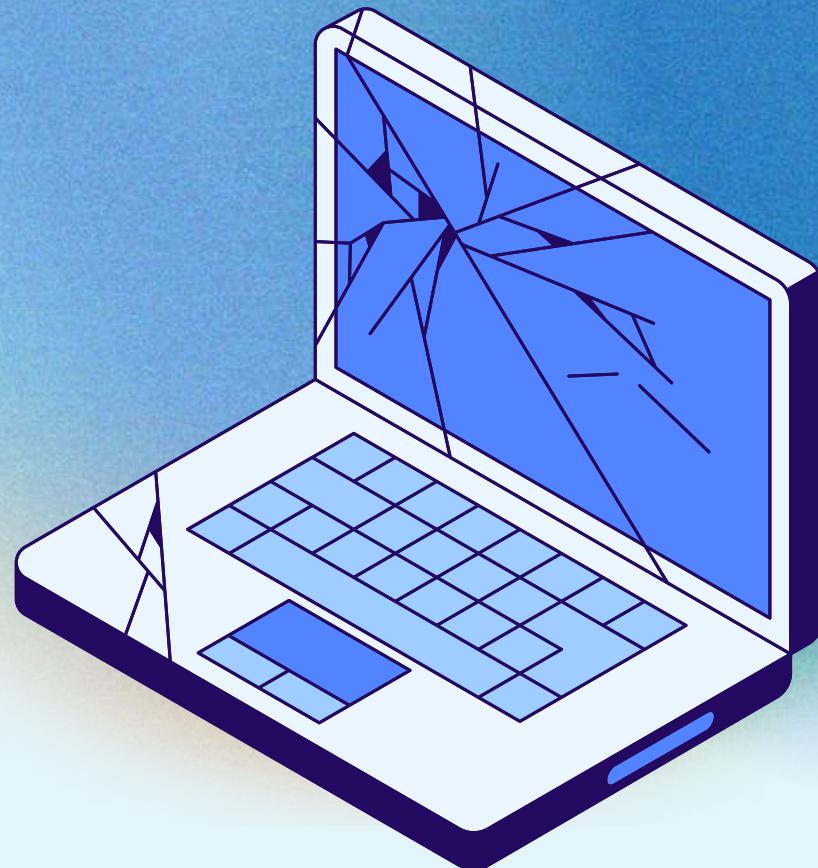
- 🌐 **Design → Development → Testing → Release**
- 🌐 **Testing ensures reliability before launch**
- 🌐 **Testing reduces risk and uncertainty**



Software Is Also a “Product”

- 🌐 Not physical, but it behaves
- 🌐 Daily dependence on software
- 🌐 Software failures can cost more than physical failures

Catastrophic Software Failures



Ariane 5 rocket explosion – type conversion error

A number was converted incorrectly (64-bit → 16-bit), causing an overflow and destroying the rocket.

Lesson: Always test data boundaries and type conversions.



Knight Capital – \$440M Loss in 45 Minutes

Old, untested code accidentally activated and executed huge unintended trades.

Lesson: Regression tests prevent old bugs from coming back.

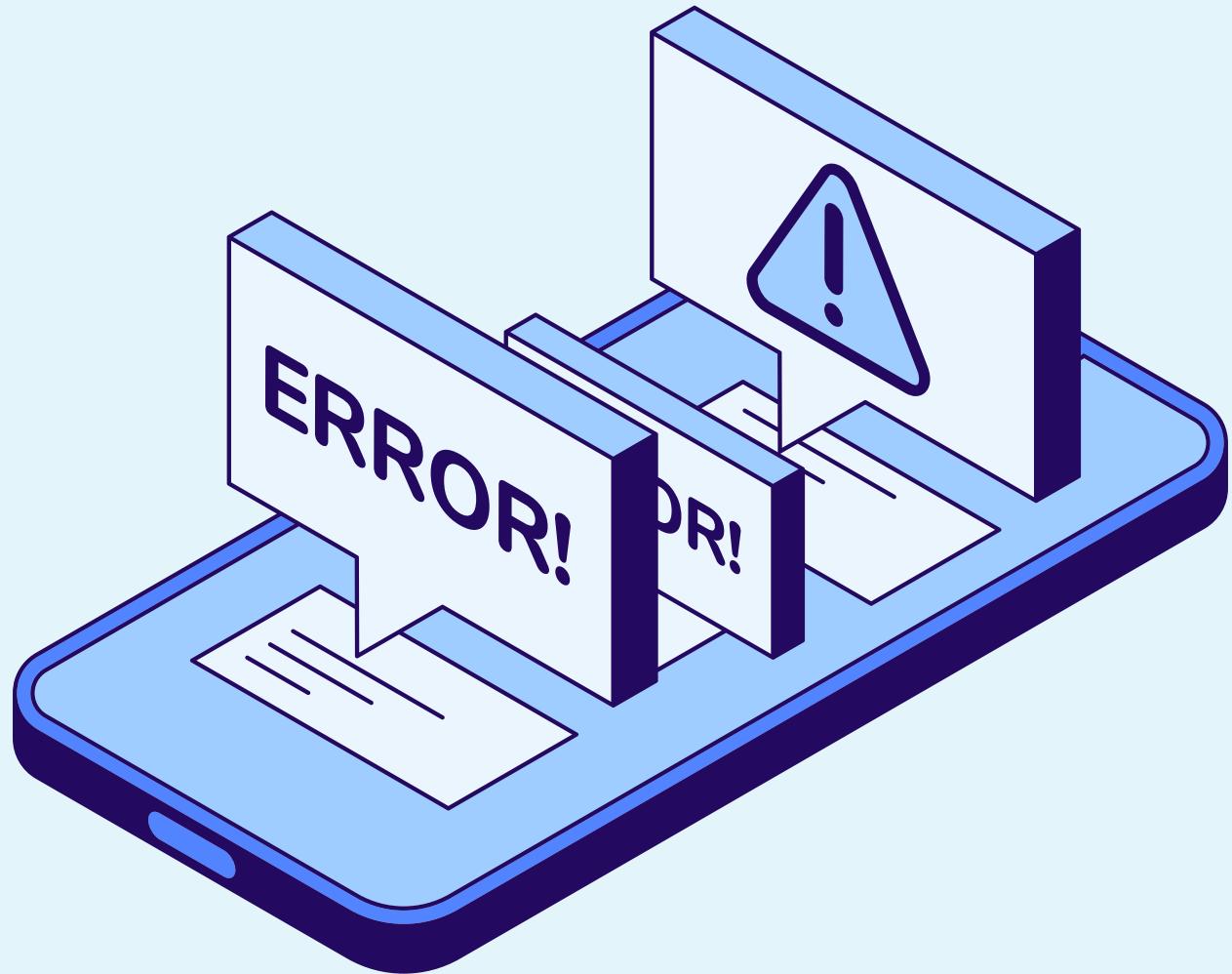


Mars Climate Orbiter – Unit Mismatch

One team used metric units, another used imperial, causing navigation failure and loss of the spacecraft.

Lesson: Test integrations carefully, especially units and shared data.

Why Is Software Error-Prone?



- 🌐 **High complexity**
- 🌐 **Dynamic behavior**
- 🌐 **Many dependencies**
- 🌐 **Error growth is exponential with project size**

Qualities of Good Software

Functional Requirements

Functional qualities describe what the software should do – the correctness of behavior.

Correctness

The software produces the right outputs for the given inputs.

It behaves exactly as the requirements specify.

Security

The system prevents unauthorized access and protects data.

Includes authentication, authorization, and safe handling of sensitive operations.

Non-Functional Requirements

Non-functional qualities describe how well the system works – performance, stability, and maintainability.

Reliability

The system runs without failures for long periods.
Consistent and dependable behavior.

Performance

The system responds quickly and uses resources efficiently.
Includes latency, throughput, and memory/CPU usage.

Maintainability

The code is easy to modify, fix, or extend.
Supports clean structure, good naming, and modular design.





Dimensions of Software Testing

Unit Testing

Tests the smallest pieces of code (functions/classes) in complete isolation.

Integration Testing

Checks that multiple components/modules work correctly together.

System Testing

Validates the entire application as a whole in a real-like environment.

Acceptance Testing

Ensures the system meets business requirements and user expectations.

Regression Testing

Verifies that new changes haven't broken previously working features.

Smoke / Sanity Testing

A quick, basic check to confirm that the system is stable enough for deeper testing.

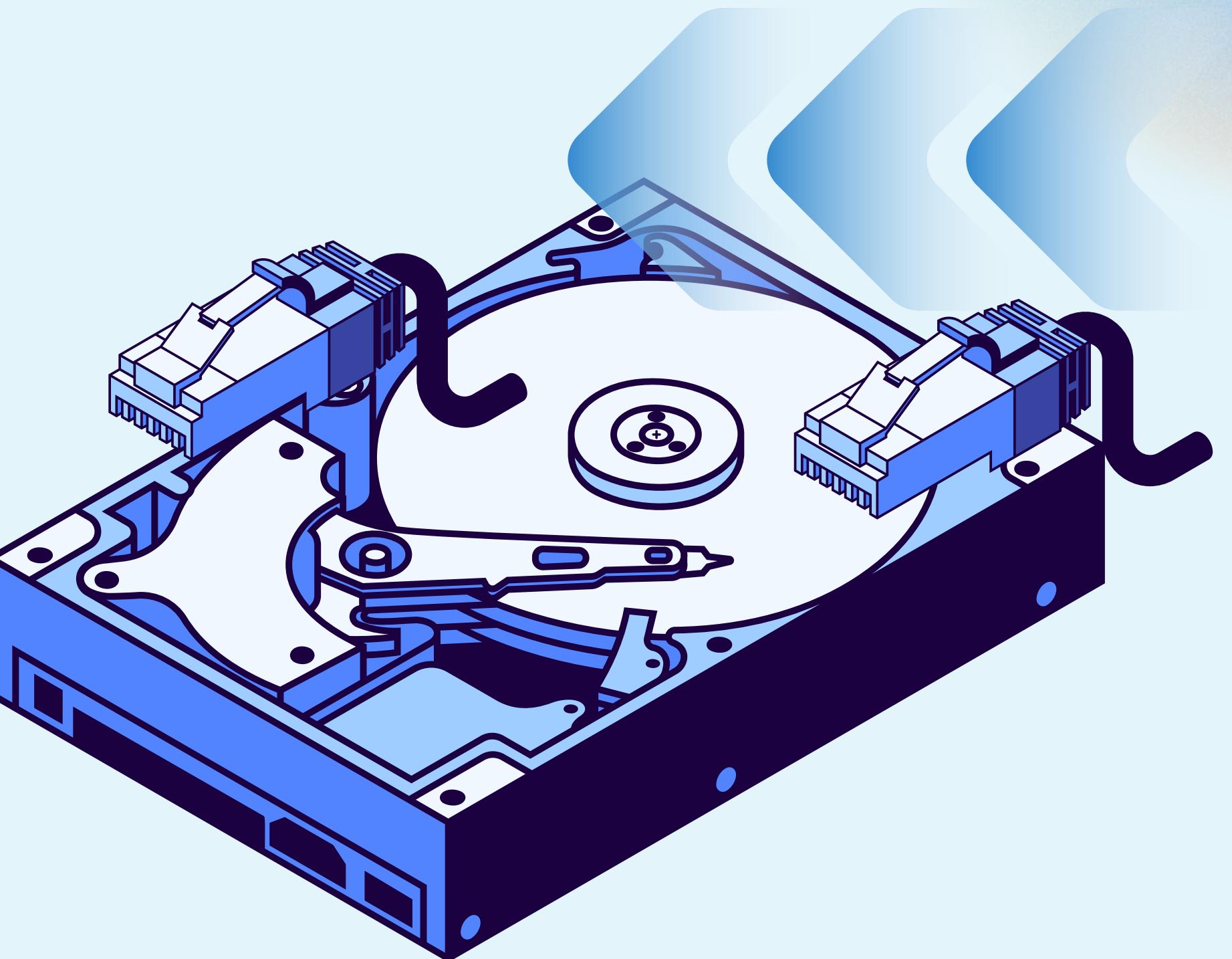
Why Developers Avoid Testing



- 🌐 **Illusion of “moving faster”**
- 🌐 **Lack of culture or skill**
- 🌐 **Fear of modifying code**
- 🌐 **Deadline pressure**

What Makes a Good Test?

- 🌐 **Clear and simple**
- 🌐 **Independent**
- 🌐 **Good coverage**
- 🌐 **Low maintenance cost**
- 🌐 **Fast execution**





Why Manual Testing Is Not Enough

- 🌐 **Not repeatable**
- 🌐 **Slow**
- 🌐 **Human error**
- 🌐 **Poor coverage**
- 🌐 **Not suitable for Agile / CI/CD**
- 🌐 **Becomes expensive as project grows**

The FIRST Principles

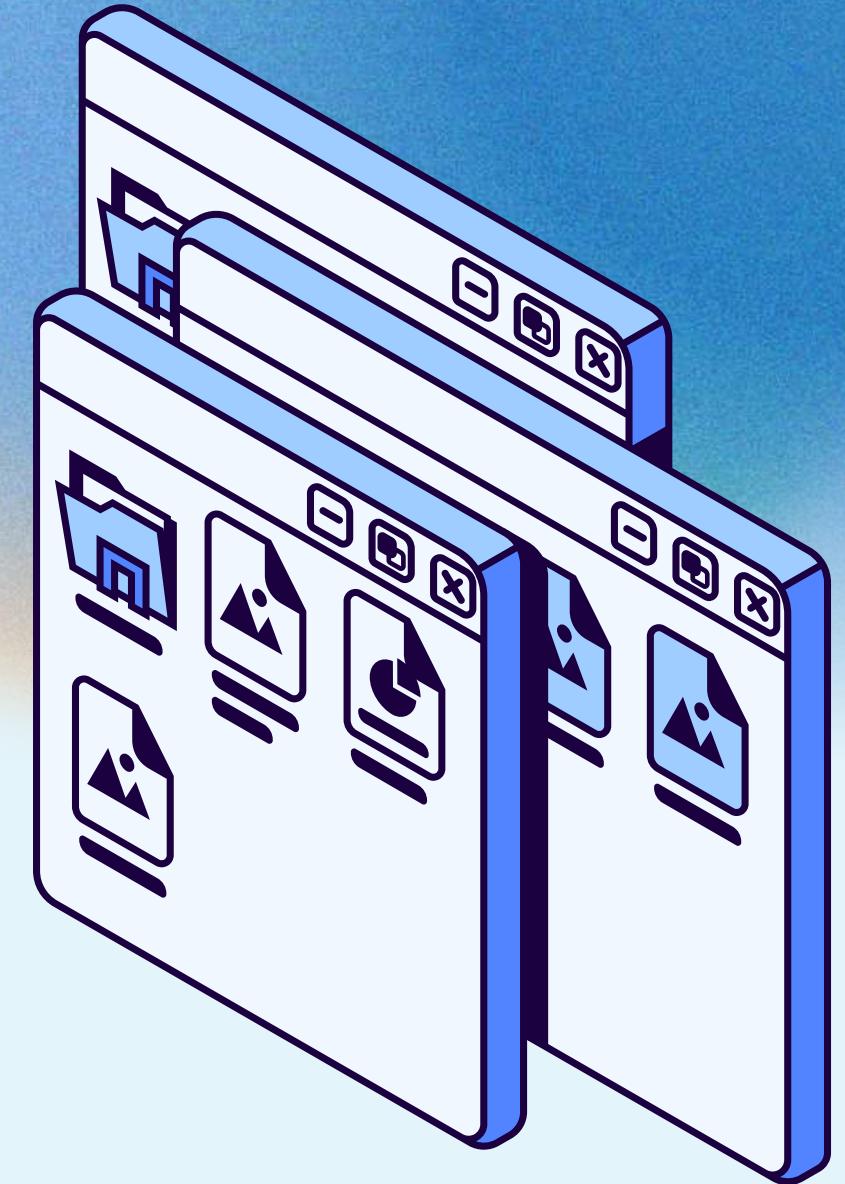
 **Fast**

 **Independent**

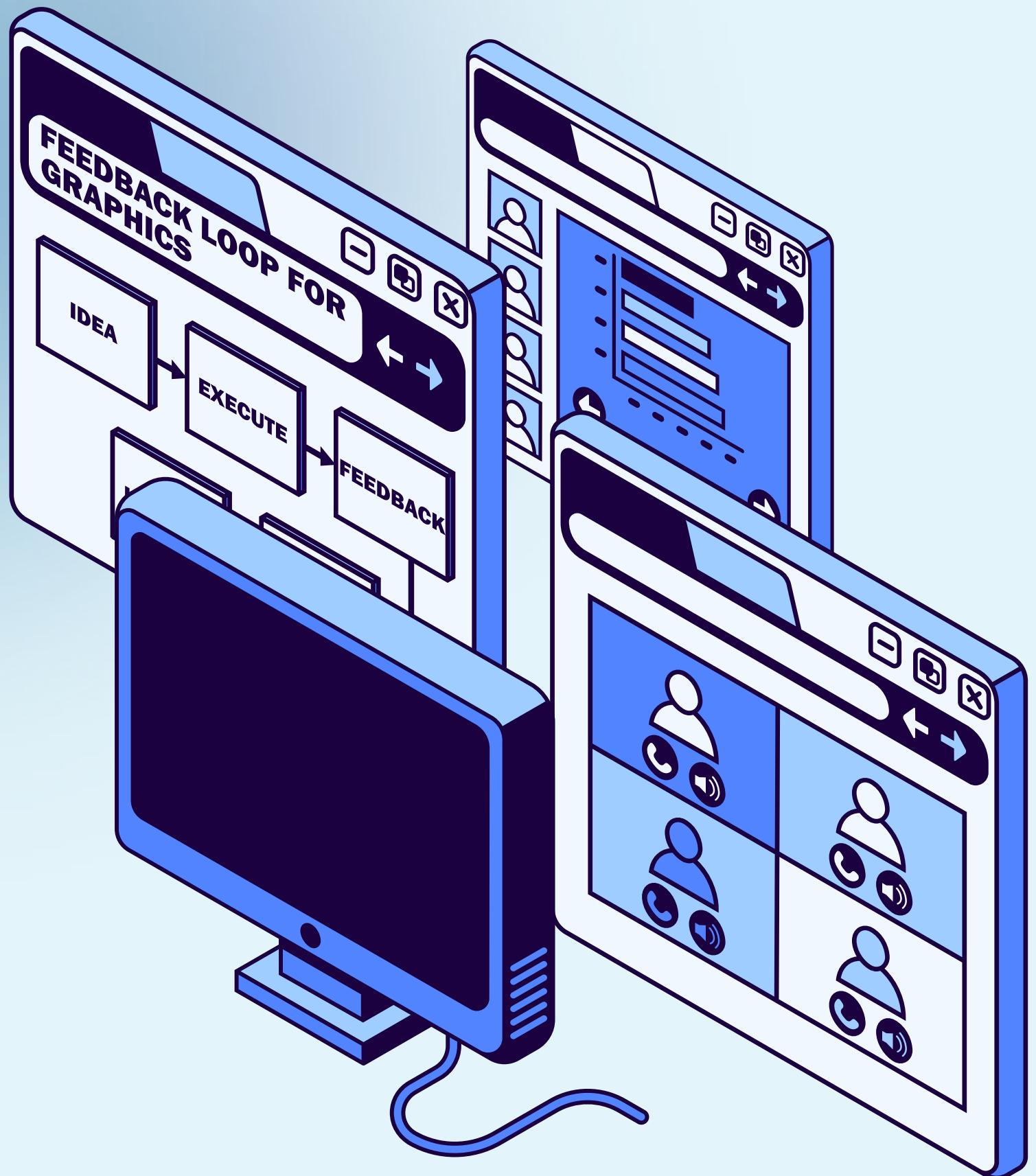
 **Repeatable**

 **Self-validating**

 **Timely**



What Is JUnit?



- 🌐 **Most popular testing framework for Java**
- 🌐 **Used to write and run unit tests**
- 🌐 **Provides annotations like @Test**
- 🌐 **Offers rich assertions for checking results**
- 🌐 **Integrates with Maven, Gradle, IntelliJ, VS Code, CI/CD**

Structure of a JUnit Test

- 🌐 **@Test annotation**
- 🌐 **Arrange → Act → Assert pattern**
- 🌐 **Clear naming**
- 🌐 **Assertions like assertEquals, assertThrows**



Test Lifecycle

🌐 **@BeforeEach**

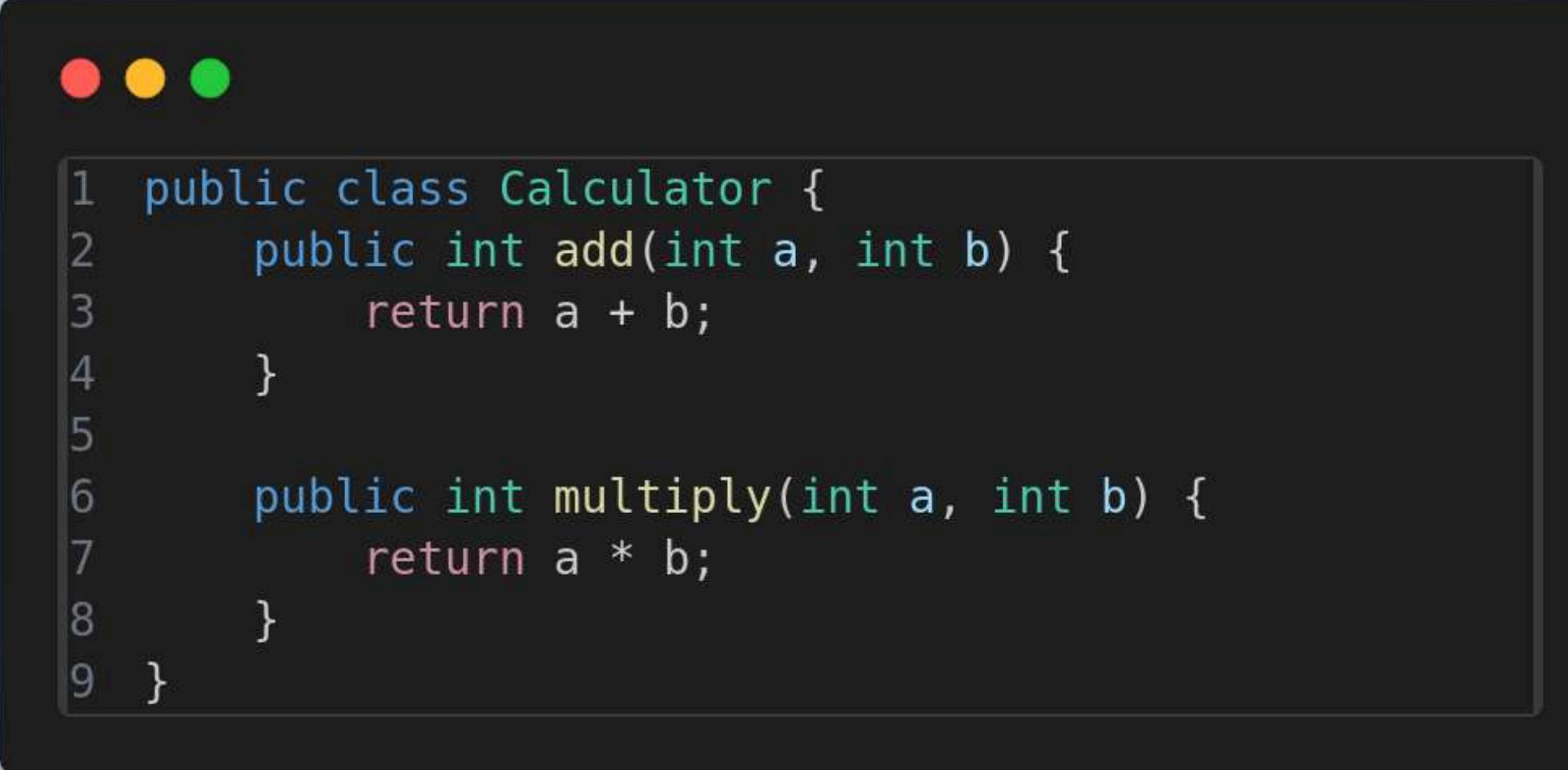
🌐 **@AfterEach**

🌐 **@BeforeAll**

🌐 **@AfterAll**



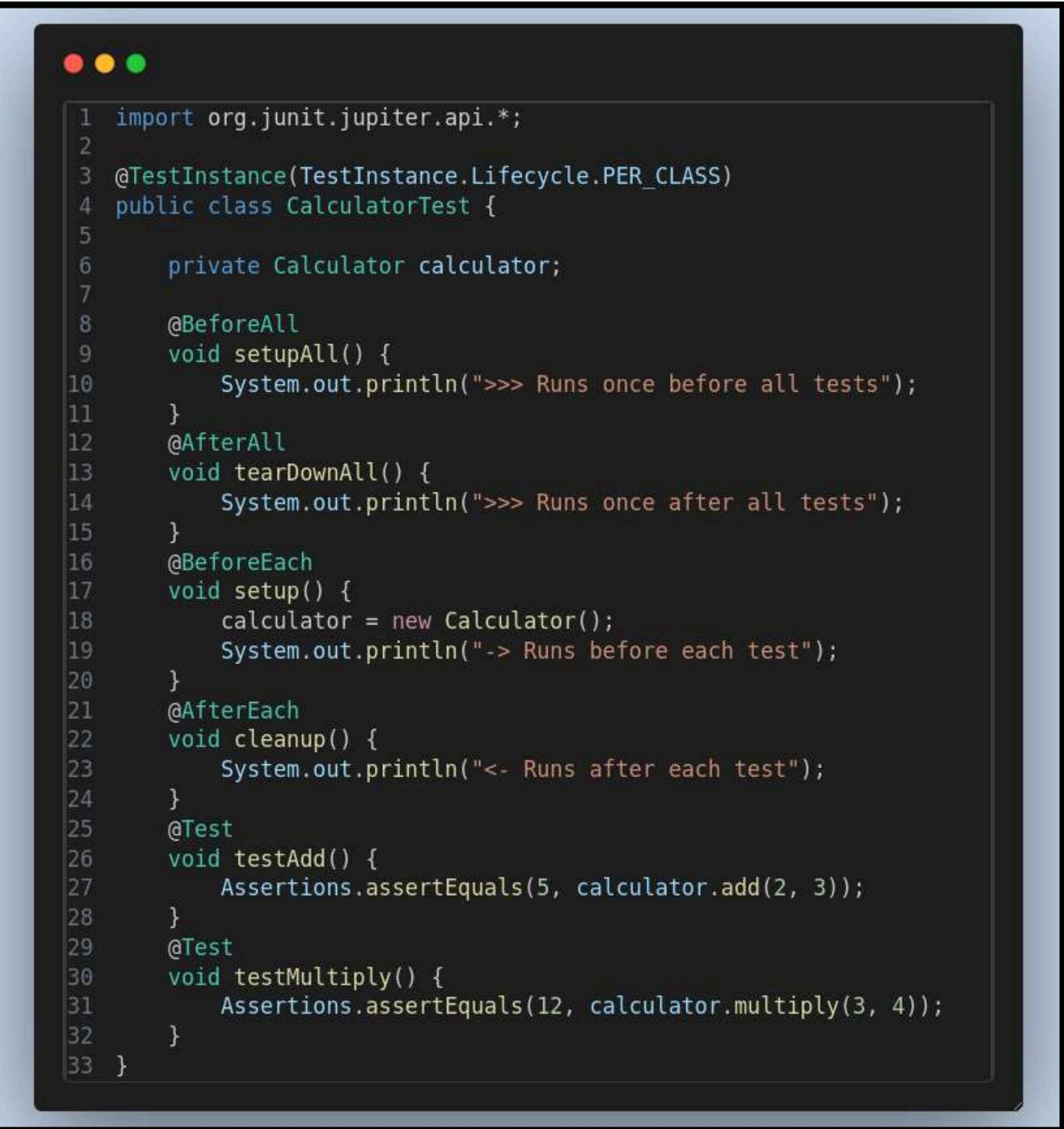
Example (source code)



The image shows a dark-themed mobile application interface, likely from an Android device, displaying Java source code. At the top left of the screen are three colored dots: red, yellow, and green. The code itself is as follows:

```
1 public class Calculator {  
2     public int add(int a, int b) {  
3         return a + b;  
4     }  
5  
6     public int multiply(int a, int b) {  
7         return a * b;  
8     }  
9 }
```

Example (test)



A screenshot of a Java code editor displaying a JUnit test class. The code uses the JUnit Jupiter API and includes lifecycle methods like @BeforeAll, @AfterAll, @BeforeEach, and @AfterEach. It also contains two test methods, testAdd() and testMultiply(), which assert the results of calculator operations.

```
1 import org.junit.jupiter.api.*;
2
3 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
4 public class CalculatorTest {
5
6     private Calculator calculator;
7
8     @BeforeAll
9     void setupAll() {
10         System.out.println(">>> Runs once before all tests");
11     }
12     @AfterAll
13     void tearDownAll() {
14         System.out.println(">>> Runs once after all tests");
15     }
16     @BeforeEach
17     void setup() {
18         calculator = new Calculator();
19         System.out.println("-> Runs before each test");
20     }
21     @AfterEach
22     void cleanup() {
23         System.out.println("<- Runs after each test");
24     }
25     @Test
26     void testAdd() {
27         Assertions.assertEquals(5, calculator.add(2, 3));
28     }
29     @Test
30     void testMultiply() {
31         Assertions.assertEquals(12, calculator.multiply(3, 4));
32     }
33 }
```

Exception Testing

- 🌐 **Use assertThrows()**
- 🌐 **Validates behavior under invalid or unexpected conditions**
- 🌐 **Ensures proper error handling**



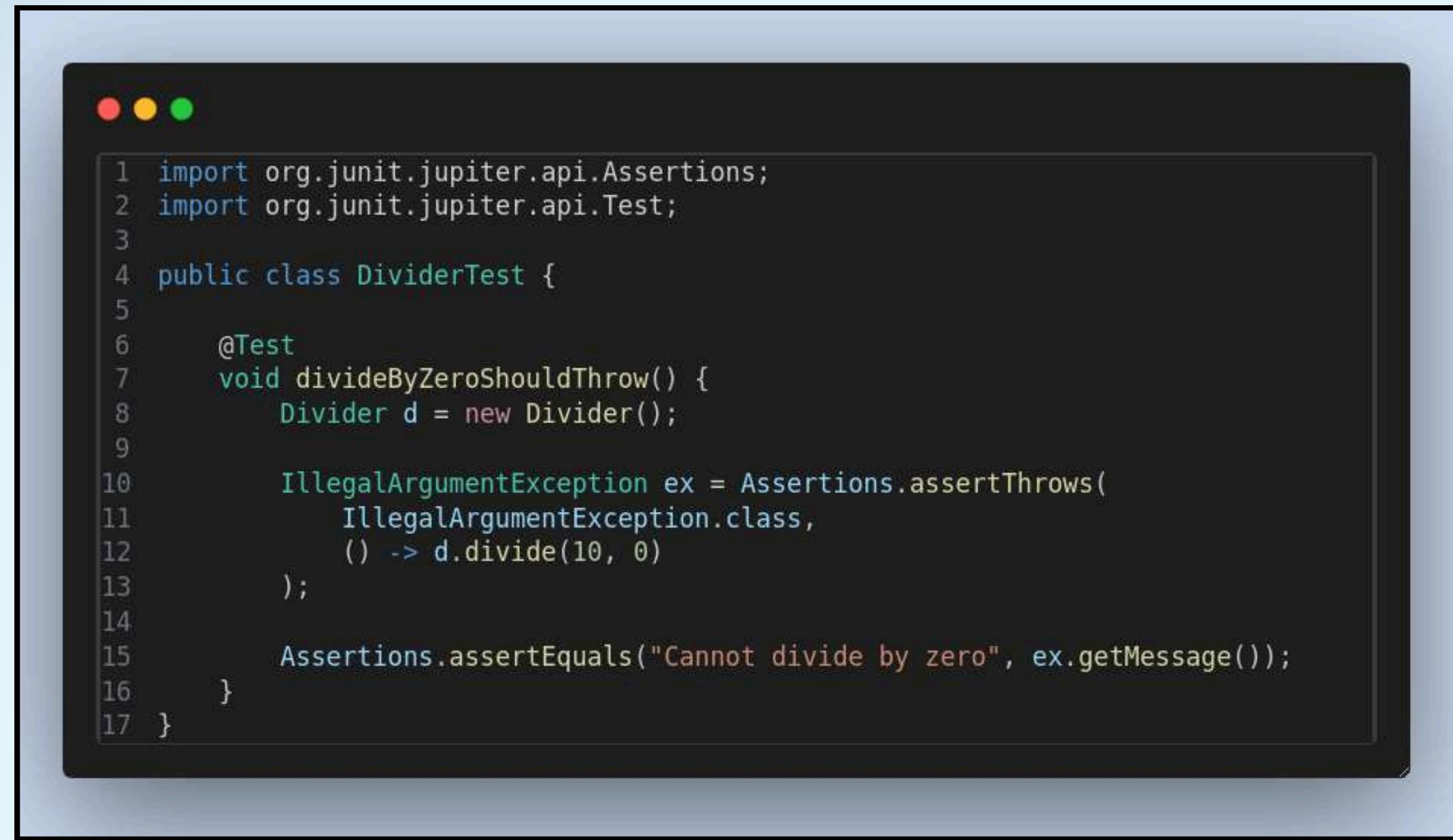
Example (source code)



A screenshot of a Java code editor showing a file named `Divider.java`. The code defines a class `Divider` with a single method `divide`. The method takes two integers, `a` and `b`, and returns their quotient. It includes a check for division by zero, throwing an `IllegalArgumentException` if `b` is zero.

```
1 public class Divider {
2     public int divide(int a, int b) {
3         if (b == 0)
4             throw new IllegalArgumentException("Cannot divide by zero");
5         return a / b;
6     }
7 }
8
```

Example (test)

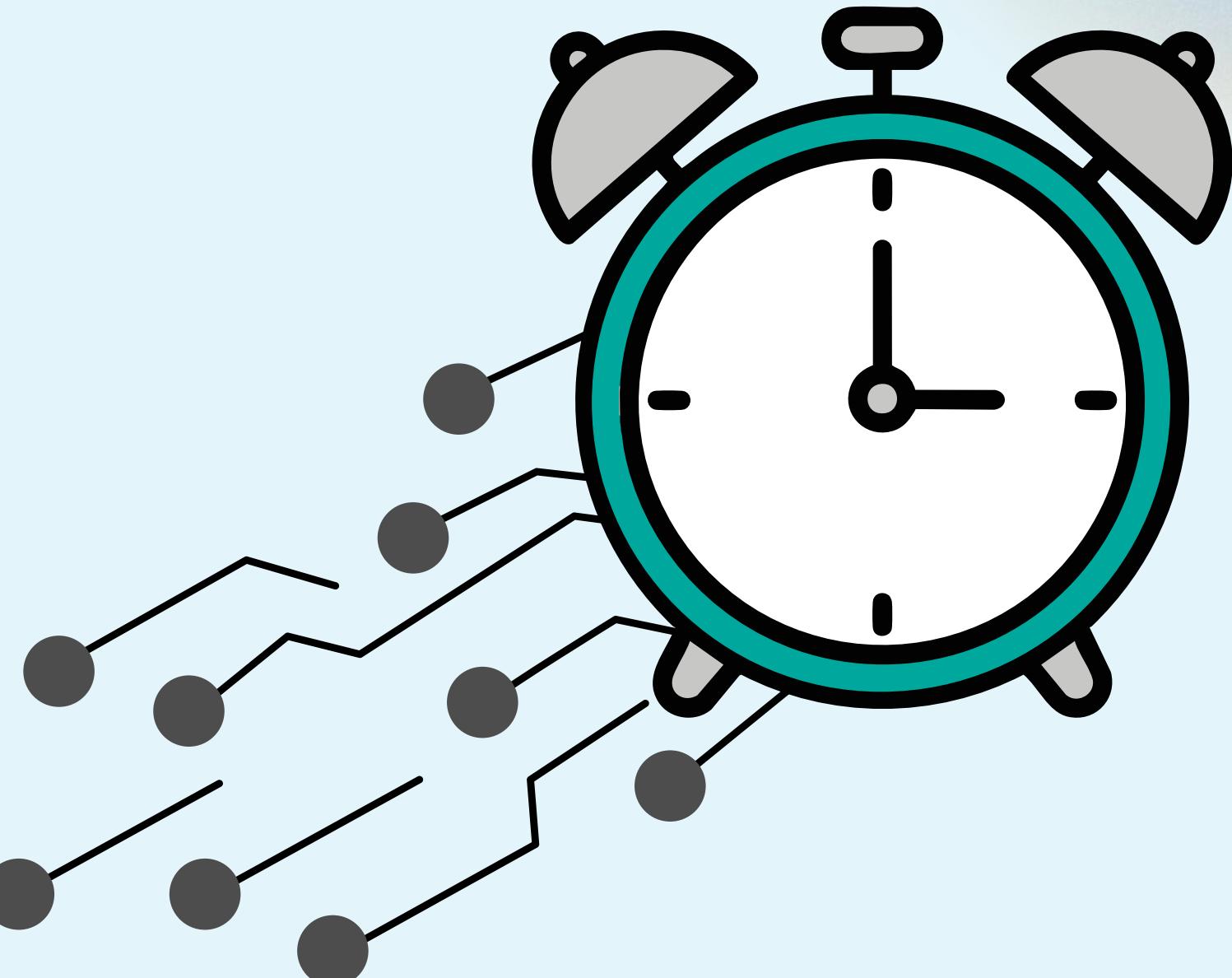


A screenshot of a terminal window with a dark background and light-colored text. The window has three colored window control buttons (red, yellow, green) at the top left. The code is a Java test class named `DividerTest` that imports `Assertions` and `Test` from `junit.jupiter`. It contains one test method, `divideByZeroShouldThrow`, which creates a `Divider` object and uses `Assertions.assertThrows` to check that dividing by zero throws an `IllegalArgumentException` with the message "Cannot divide by zero".

```
1 import org.junit.jupiter.api.Assertions;
2 import org.junit.jupiter.api.Test;
3
4 public class DividerTest {
5
6     @Test
7     void divideByZeroShouldThrow() {
8         Divider d = new Divider();
9
10        IllegalArgumentException ex = Assertions.assertThrows(
11            IllegalArgumentException.class,
12            () -> d.divide(10, 0)
13        );
14
15        Assertions.assertEquals("Cannot divide by zero", ex.getMessage());
16    }
17 }
```

Timeout Testing

- 🌐 **Use assertTimeout()**
- 🌐 **Prevents hanging tests**
- 🌐 **Checks basic performance expectations**

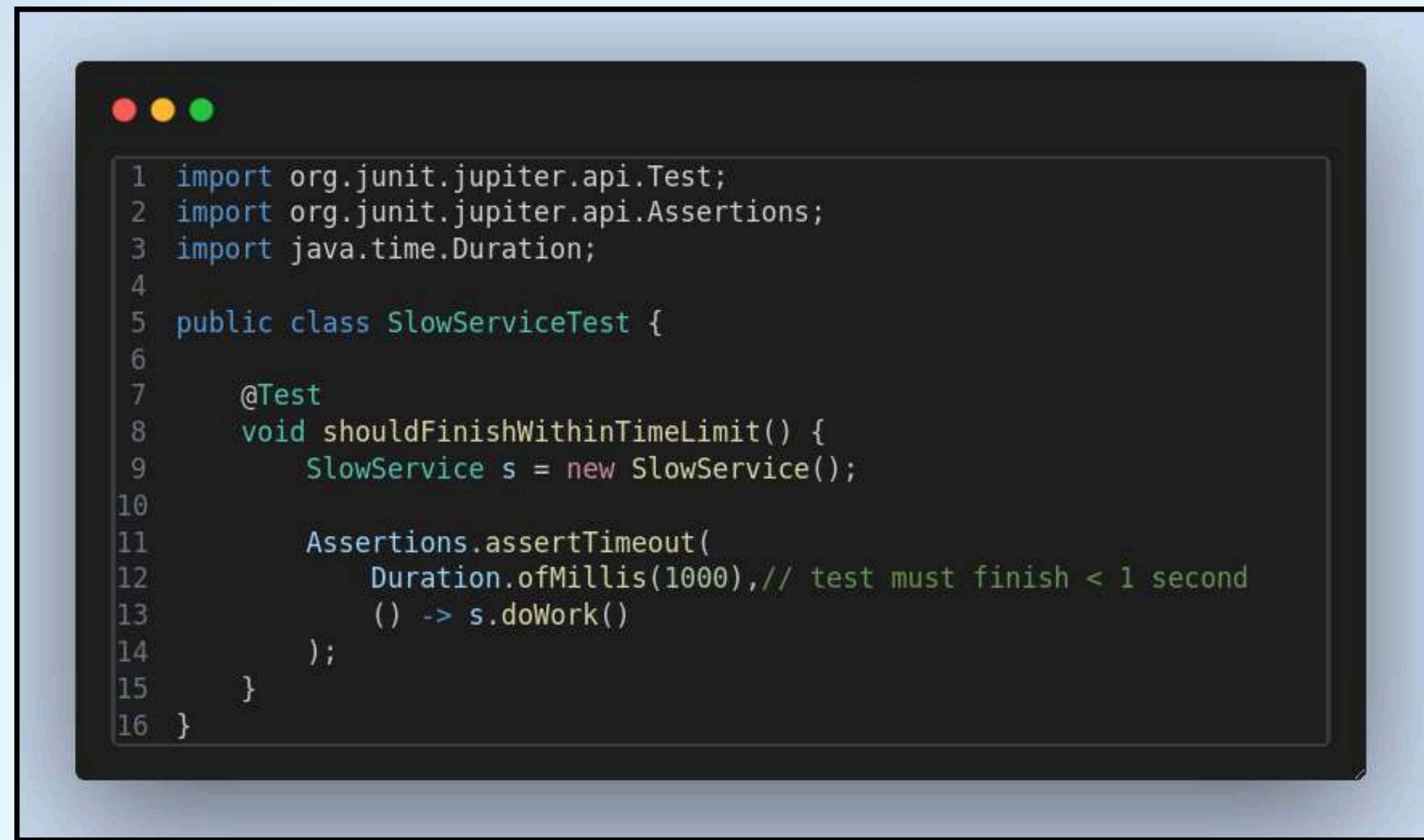


Example (source code)



```
1 public class SlowService {
2     public void doWork() {
3         try {
4             Thread.sleep(500); // simulates slow task
5         } catch (InterruptedException e) {
6             throw new RuntimeException(e);
7         }
8     }
9 }
```

Example (test)



```
1 import org.junit.jupiter.api.Test;
2 import org.junit.jupiter.api.Assertions;
3 import java.time.Duration;
4
5 public class SlowServiceTest {
6
7     @Test
8     void shouldFinishWithinTimeLimit() {
9         SlowService s = new SlowService();
10
11         Assertions.assertTimeout(
12             Duration.ofMillis(1000), // test must finish < 1 second
13             () -> s.doWork()
14         );
15     }
16 }
```

Test Categorization

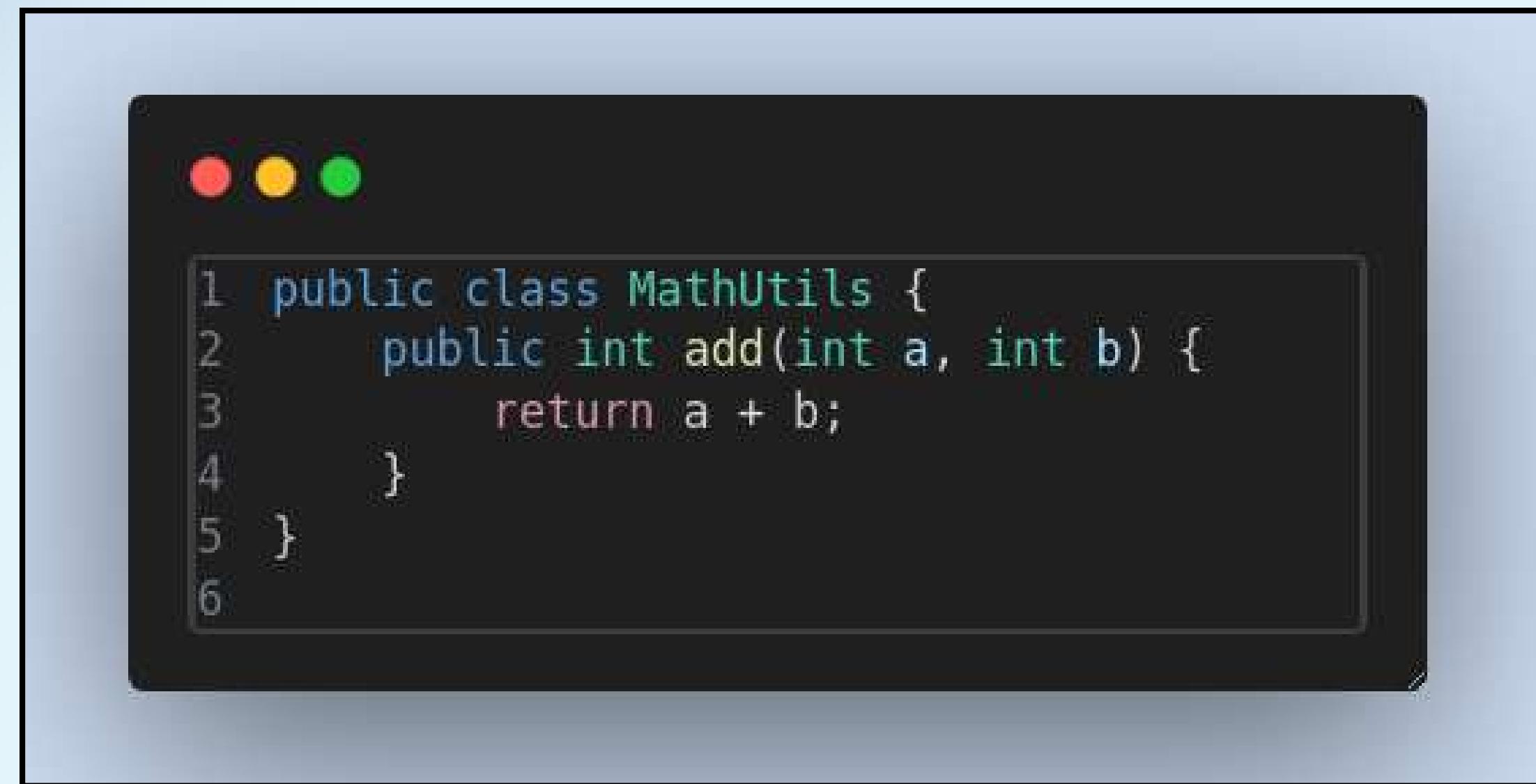
- 🌐 **Use @Tag**

- 🌐 **Group tests**

- 🌐 **Run specific test groups in CI/CD**



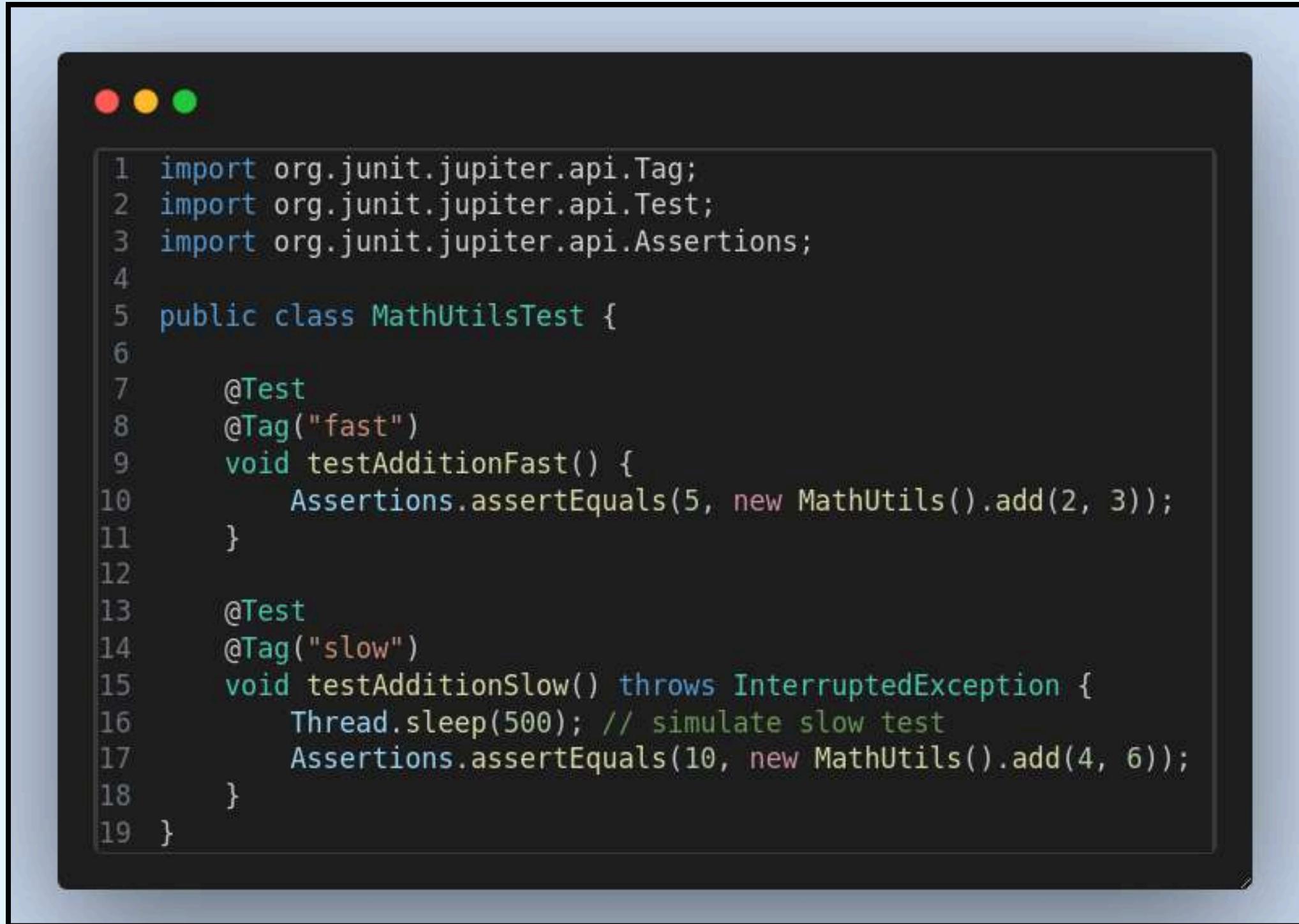
Example (source code)



A screenshot of a mobile application interface, likely a code editor or terminal window. The background is dark gray. At the top left, there are three small colored circles: red, yellow, and green. Below them is a black rectangular area containing white text representing Java source code:

```
1 public class MathUtils {  
2     public int add(int a, int b) {  
3         return a + b;  
4     }  
5 }  
6
```

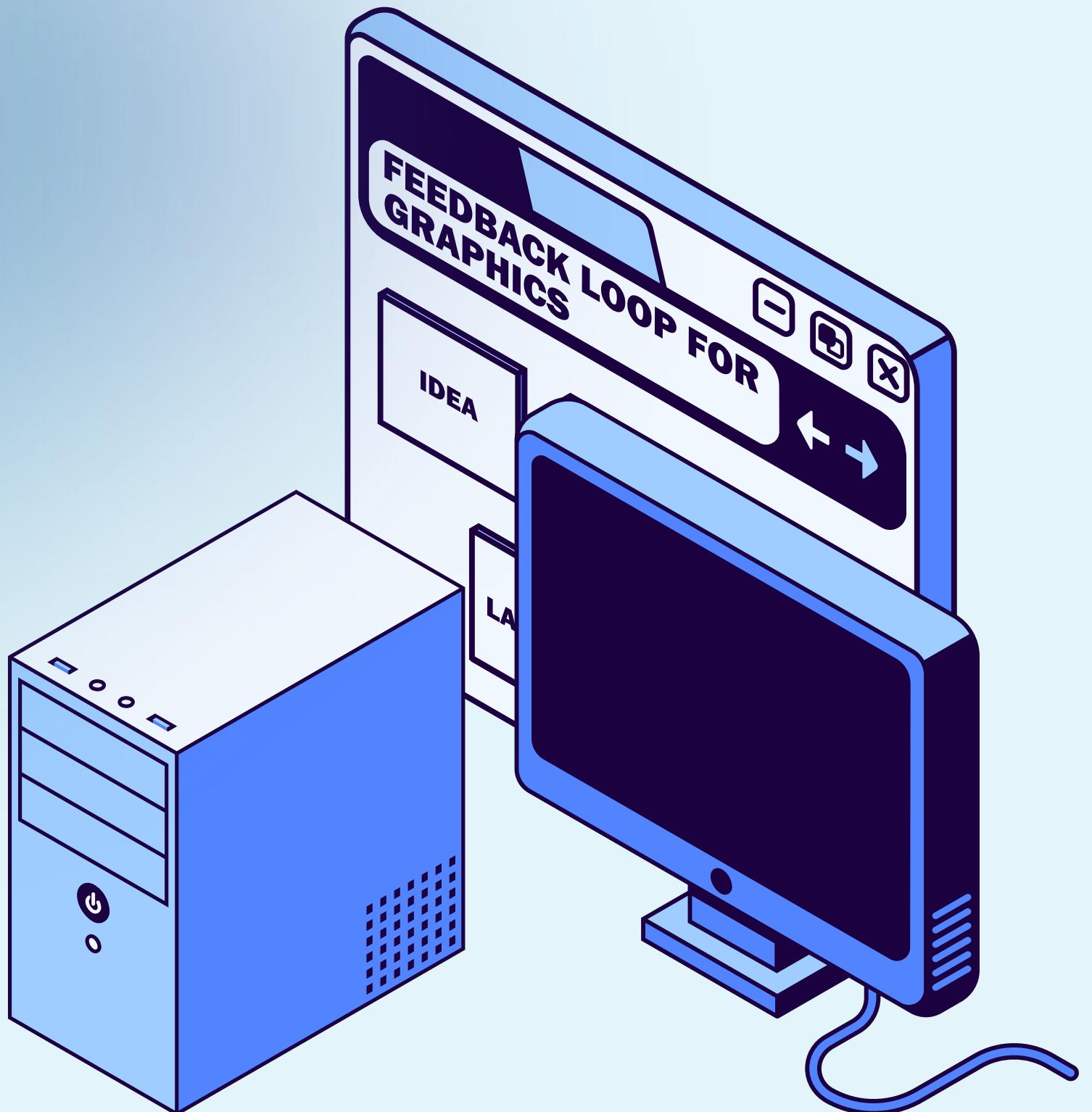
Example (test)



A screenshot of a terminal window displaying Java code. The code is a test class named `MathUtilsTest` using the JUnit Jupiter API. It contains two test methods: `testAdditionFast` and `testAdditionSlow`. The `testAdditionSlow` method includes a `Thread.sleep(500)` call to simulate a slow test.

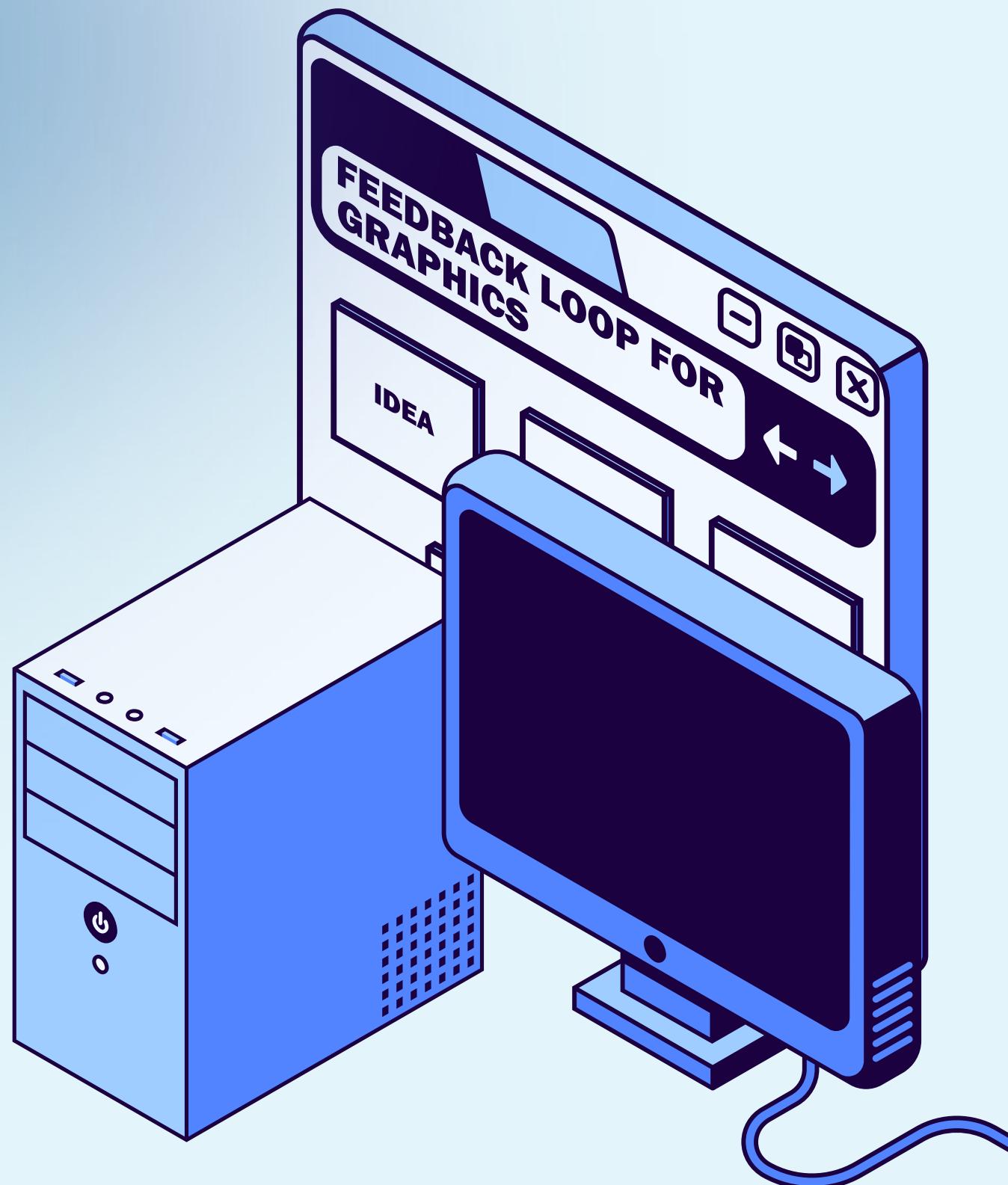
```
1 import org.junit.jupiter.api.Tag;
2 import org.junit.jupiter.api.Test;
3 import org.junit.jupiter.api.Assertions;
4
5 public class MathUtilsTest {
6
7     @Test
8     @Tag("fast")
9     void testAdditionFast() {
10         Assertions.assertEquals(5, new MathUtils().add(2, 3));
11     }
12
13     @Test
14     @Tag("slow")
15     void testAdditionSlow() throws InterruptedException {
16         Thread.sleep(500); // simulate slow test
17         Assertions.assertEquals(10, new MathUtils().add(4, 6));
18     }
19 }
```

Setting Up JUnit in Maven



- 🌐 Add dependency to pom.xml
- 🌐 Use maven-surefire-plugin
- 🌐 Provides annotations like @Test
- 🌐 Standard directories:
 - src/main/java
 - src/test/java

Setting Up JUnit in Gradle



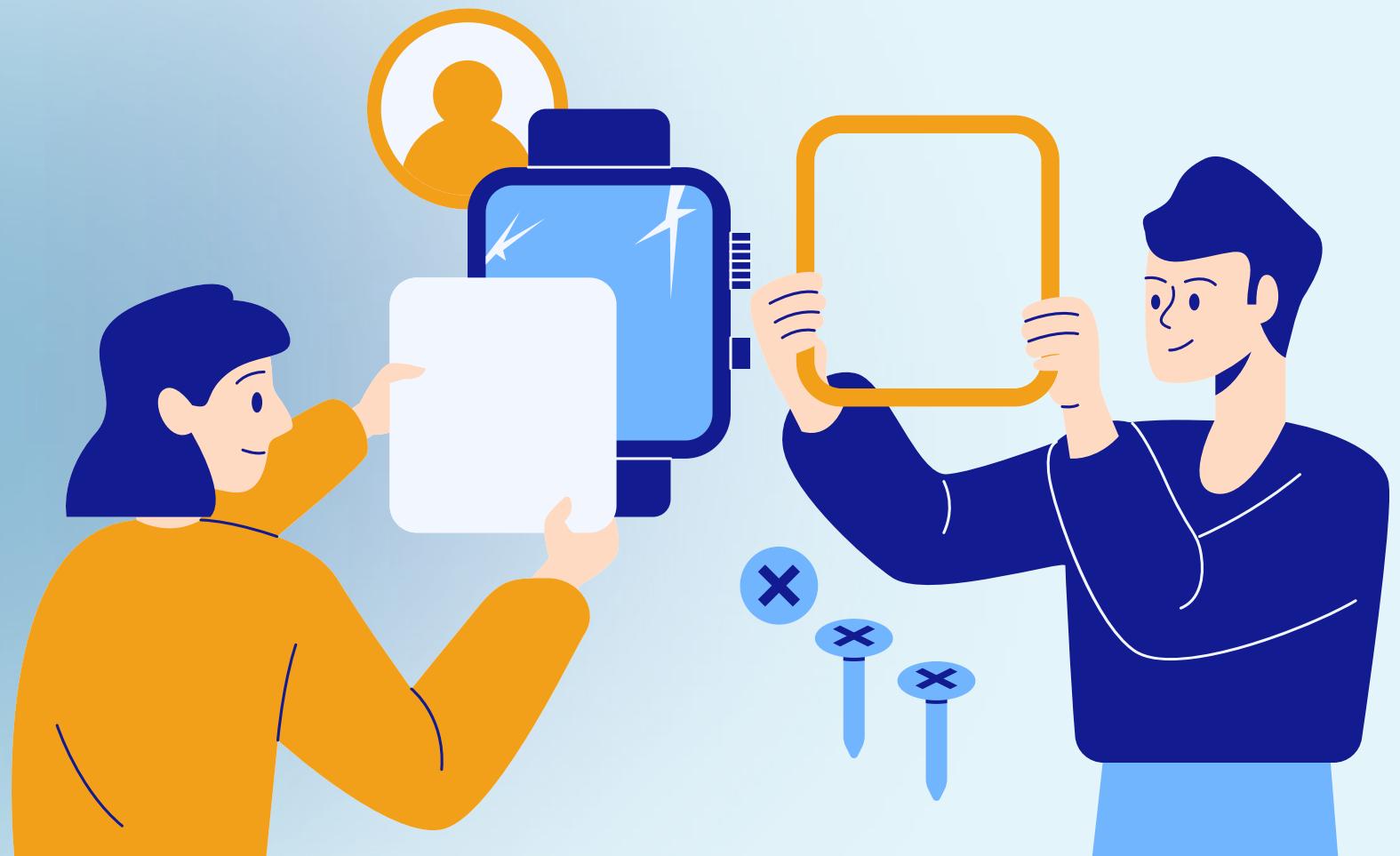
- 🌐 **Use `testImplementation`**
- 🌐 **Run tests with `Gradle test runner`**
- 🌐 **Configure source sets if needed**

Java Tips for Better Testing

- 🌐 **Use immutability**
- 🌐 **Handle exceptions properly**
- 🌐 **Implement equals and hashCode correctly**
- 🌐 **Be careful with static methods**
- 🌐 **Use dependency injection**



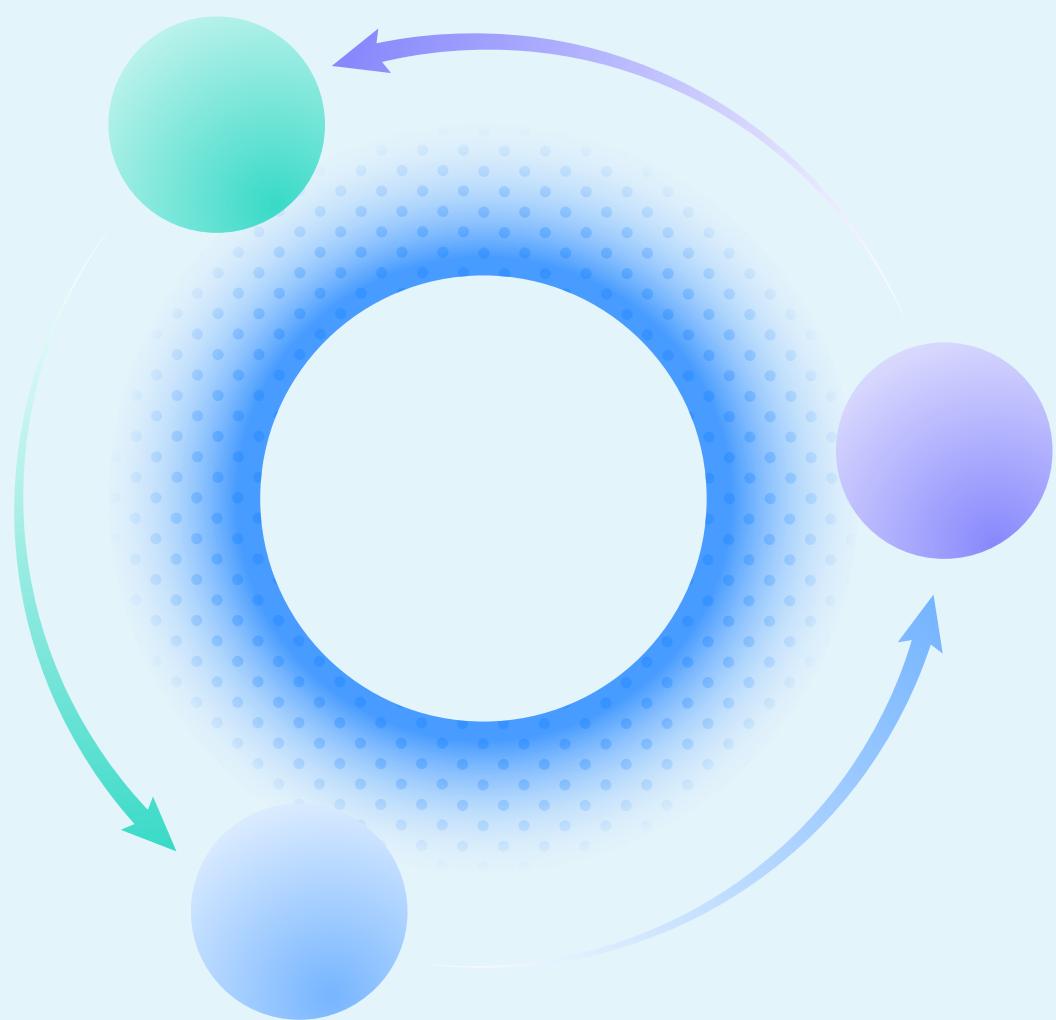
What Is TDD?



Test-Driven Development (TDD)

- 🌐 A development method where tests are written before writing code
- 🌐 Ensures clarity of requirements
- 🌐 Helps avoid over-engineering
- 🌐 Produces clean, modular, testable code

The TDD Cycle (Red → Green → Refactor)



- 🌐 **Red:** write a failing test
- 🌐 **Green:** write minimum code to make the test pass
- 🌐 **Refactor:** improve the code while keeping tests green
- 🌐 **Repeat for every feature**

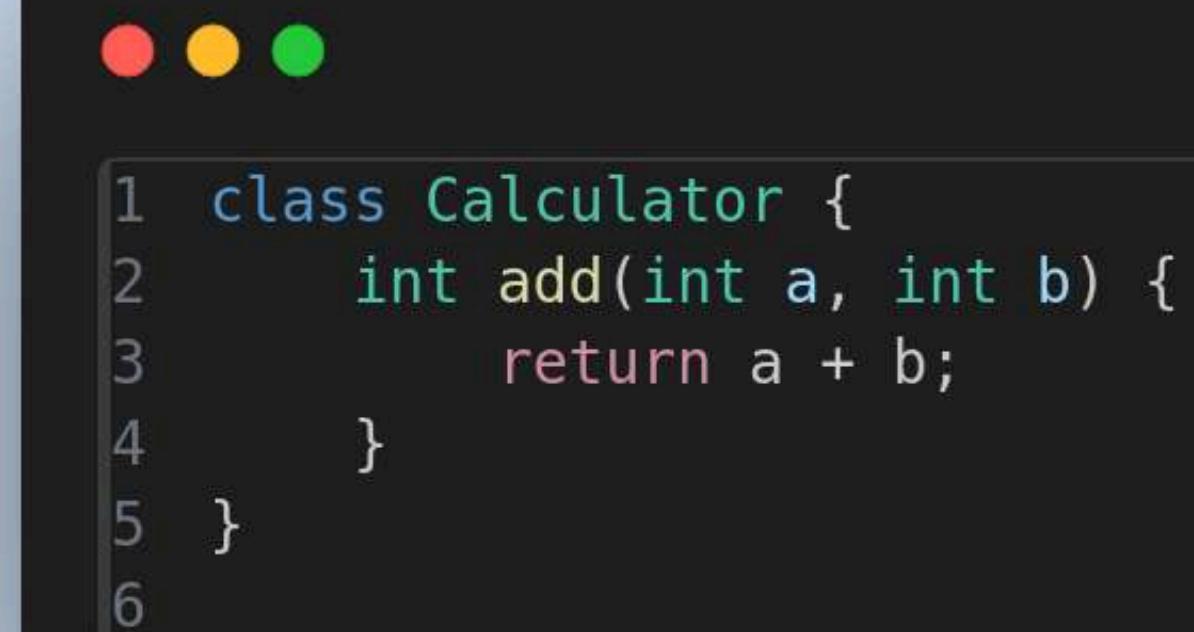
Simple TDD Example (Java + JUnit)

Test (write first):

```
1 import org.junit.jupiter.api.Test;  
2 import static org.junit.jupiter.api.Assertions.*;  
3  
4 class CalculatorTest {  
5     @Test  
6     void add_twoNumbers_returnsSum() {  
7         Calculator c = new Calculator();  
8         assertEquals(7, c.add(3, 4));  
9     }  
10 }
```

Simple TDD Example (Java + JUnit)

**Code
(written after test fails)**

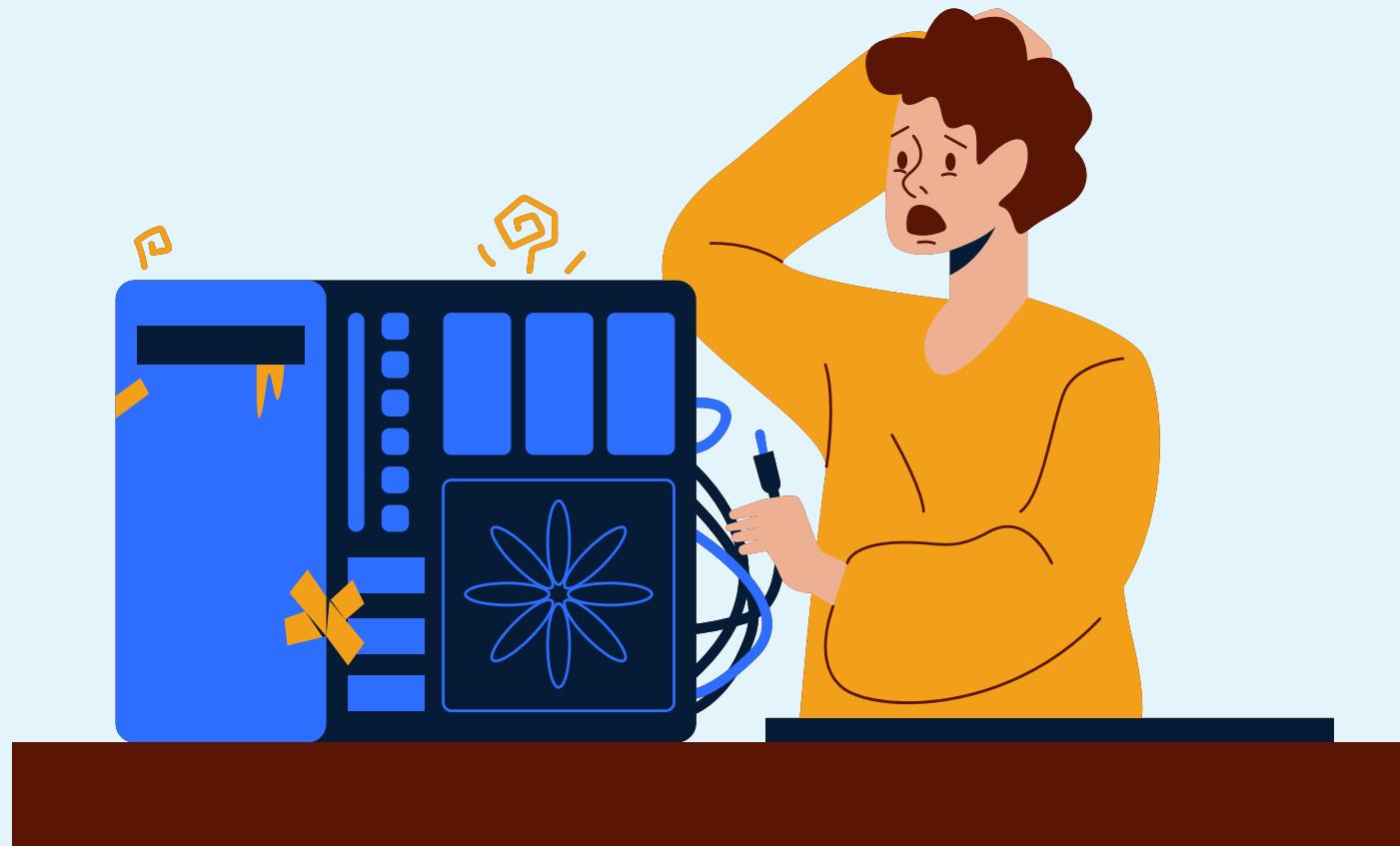


A terminal window with a black background and a light gray border. At the top, there are three colored dots: red, yellow, and green. Below them, the Java code for a 'Calculator' class is displayed:

```
1 class Calculator {  
2     int add(int a, int b) {  
3         return a + b;  
4     }  
5 }  
6
```

Why TDD Works

- 🌐 Forces good design early
- 🌐 Gives immediate feedback
- 🌐 Implements equals and hashCode correctly
- 🌐 Prevents fear of changing code
- 🌐 Creates a safety net for refactoring
- 🌐 Reduces debugging time dramatically



Conclusion



- 🌐 **Unit testing builds confidence and catches bugs early**
- 🌐 **JUnit makes testing simple, fast, and structured**
- 🌐 **TDD improves design, keeps code clean, and reduces rework**
- 🌐 **Small, focused tests lead to reliable and maintainable software**
- 🌐 **Testing is not extra work – it's part of writing good code**

