

# Testing your code

# Why do we write tests?

It is not just about finding bugs. Testing provides:

- **Ability to verify** code even if the overall application or feature isn't complete.
- **Confidence**: Makes it easier to refactor and change the code with a higher degree of confidence.
- **Documentation**: Tests show the expected behaviour of your code.
- **Guides Design**: Hard-to-test code is often poorly designed (tightly coupled).
- **Cost Savings**: Bugs found in development are cheaper to fix than bugs in production.

Is writing tests a waste of time?

- **The Myth**: "I don't have time to write tests; I need to ship features."
- **The Reality**: Manual testing. A recurring cost. You must manually re-check features for every change.
- **Unit Testing**: A one-time investment. You write it once, and this case is checked forever.

# The Testing Pyramid

We should organize our tests based on speed and cost.

1. **Unit Tests (Base):** Fast, isolated, cheap. Covers individual functions. **70% of tests.**
2. **Integration Tests (Middle):** Checks how modules work together. Slower. **20% of tests.**
3. **E2E / UI Tests (Top):** Simulates user behavior. Slow and brittle. **10% of tests.**

**Goal:** Push tests down the pyramid. If you can test it with a Unit Test, don't use an E2E test.

# Anatomy of a Unit Test

Follow the AAA pattern for clarity:

1. **Arrange:** Set up the data and objects needed.
2. **Act:** Call the method you want to test.
3. **Assert:** Verify that the result matches your expectation.

```
@Test
fun `addition adds two fractions correctly`() {
    // Arrange
    val f1 = Fraction(1, 2)
    val f2 = Fraction(1, 4)

    // Act
    val result = f1 + f2

    // Assert
    assertEquals(Fraction(3, 4), result)
}
```

## JUnit 5: The Test Lifecycle

Tests often require preparation (Setup) and cleanup (Teardown) so they don't interfere with each other. JUnit provides annotations to control when code runs.

- `@BeforeEach` : Runs **before every** `@Test`. Great for initializing fresh objects (Setup).
- `@AfterEach` : Runs **after every** `@Test`. Used to clean up resources, close files, or reset states (Teardown).
- `@BeforeAll` / `@AfterAll` : Runs **once** for the entire test class. (Must be placed in a `companion object` in Kotlin).

# Setup & Teardown in Action

Here is how we use the lifecycle to ensure every test gets a fresh, isolated environment:

```
class FractionTest {
    private lateinit var fraction: Fraction

    @BeforeEach
    fun setUp() {
        // Runs before EACH test. We get a fresh instance!
        fraction = Fraction(1, 2)
    }

    @AfterEach
    fun tearDown() {
        // Runs after EACH test.
        // (Usually more useful for closing databases or files)
    }

    @Test
    fun testSomething() { /* ... */ }
}
```

## Core Assert Functions (JUnit 5)

Assertions are how we verify that our code did what we expected. If an assertion fails, the test fails.

- `assertEquals(expected, actual)` : Checks if two values are equal. (*Note: For custom objects, you must override the `equals()` method in your class!*)
- `assertNotEquals(unexpected, actual)` : Checks if values are different.
- `assertTrue(condition)` / `assertFalse(condition)` : Checks boolean conditions.
- `assertNull(actual)` / `assertNotNull(actual)` : Checks for nullability.

# Testing for Exceptions

Sometimes, we *expect* our code to fail (e.g., passing a `0` denominator). We must test that the correct exception is thrown using `assertThrows`.

```
@Test
fun `fraction throws exception on zero denominator`() {
    // Act & Assert
    val exception = assertThrows<IllegalArgumentException> {
        Fraction(1, 0)
    }

    // Optional: Check the exact error message
    assertEquals("Denominator cannot be zero", exception.message)
}
```

## Best Practices: F.I.R.S.T. Principles

Good unit tests follow these five rules:

- **F - Fast:** Tests should run in milliseconds. If they are slow, developers won't run them.
- **I - Isolated:** One test should not depend on the result of another.
- **R - Repeatable:** Tests should produce the same result every time (no random failures).
- **S - Self-Validating:** The test should clearly say "Pass" or "Fail" without manual checking.
- **T - Timely:** Write tests *before* or *during* coding, not months later.

# **Test Driven Development (TDD)**

A disciplined approach to writing software:

1. **RED**: Write a failing test for a small feature.
2. **GREEN**: Write just enough code to make the test pass.
3. **REFACTOR**: Clean up the code while keeping the test passing.

*Cycle: Red -> Green -> Refactor*

## Common Pitfalls to Avoid

- **Testing Implementation Details:** Test *what* the code does, not *how* it does it.
- *Bad:* `assert(fraction.privateVariable == 5)`
- *Good:* `assert(fraction.getValue() == 5)`
- **Logic in Tests:** Avoid `if` statements or loops in your test code. Tests should be linear and simple.

# Summary

1. Tests are for **confidence** and **design**.
2. Stick to the **AAA** pattern.
3. Keep tests **Fast** and **Independent**.
4. Cover **Edge Cases** (like dividing by zero!).