1) What JUnit is and how JUnit 5 is structured

JUnit is the de-facto Java testing framework.

JUnit 5 = 3 parts:

1. **JUnit Platform** (test discovery + execution engine)
2. **JUnit Jupiter** (annotations + API you write tests with: `@Test`, `@BeforeEach`, etc.)
3. **JUnit Vintage** (optional: run JUnit 3/4 tests)

In Maven, you typically use **JUnit Jupiter** + Surefire plugin.

---

2) Create a Maven project

Option A: Using IntelliJ / Eclipse

Create **Maven** project, GroupId + ArtifactId, packaging `jar`.

Option B: Using Maven CLI (quick)

```
mvn -q archetype:generate \
  -DgroupId=com.example \
  -DartifactId=junit-notes \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

Then open the project.

Standard Maven structure

```
junit-notes/
  pom.xml
  src/
    main/java/...
    test/java/...
```

---

3) Add dependencies + plugins in `pom.xml` (JUnit 5)

Use this `pom.xml` baseline.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>junit-notes</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <junit.jupiter.version>5.11.0</junit.jupiter.version>
    <mockito.version>5.13.0</mockito.version>
    <assertj.version>3.26.3</assertj.version>
  </properties>

  <dependencies>
    <!-- JUnit 5 (Jupiter) -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>${junit.jupiter.version}</version>
      <scope>test</scope>
    </dependency>

    <!-- Parameterized tests -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <version>${junit.jupiter.version}</version>
      <scope>test</scope>
    </dependency>

    <!-- AssertJ for fluent assertions (optional but practical) -->
    <dependency>
```

```
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>${assertj.version}</version>
      <scope>test</scope>
    </dependency>

    <!-- Mockito for mocking dependencies -->
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>${mockito.version}</version>
      <scope>test</scope>
    </dependency>

    <!-- Mockito + JUnit 5 integration -->
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-junit-jupiter</artifactId>
      <version>${mockito.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- Run unit tests -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.5.2</version>
        <configuration>
          <useModulePath>false</useModulePath>
        </configuration>
      </plugin>

      <!-- Run integration tests (optional setup; explained later)
-->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>3.5.2</version>
```

```
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Run tests

```
mvn test
```

---

4) A small “production” example to test

Create these files under `src/main/java`.

4.1 Domain: `Money`

```
package com.example.domain;

import java.math.BigDecimal;
import java.util.Objects;

public final class Money {
    private final BigDecimal amount;

    public Money(BigDecimal amount) {
        if (amount == null) throw new
IllegalArgumentException("amount must not be null");
        this.amount = amount;
    }

    public BigDecimal amount() {
        return amount;
    }
```

```
    public Money add(Money other) {
        if (other == null) throw new IllegalArgumentException("other
must not be null");
        return new Money(this.amount.add(other.amount));
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Money)) return false;
        Money money = (Money) o;
        return amount.compareTo(money.amount) == 0; // ignore scale
differences (10.0 == 10.00)
    }

    @Override
    public int hashCode() {
        return Objects.hash(amount.stripTrailingZeros());
    }

    @Override
    public String toString() {
        return "Money{" + amount + '}';
    }
}
```

4.2 Service: `DiscountService`

```
package com.example.service;

import com.example.domain.Money;

import java.math.BigDecimal;

public class DiscountService {

    // Simple rule: if amount >= 1000 => 10% discount else 0
    public Money applyDiscount(Money original) {
        if (original == null) throw new
IllegalArgumentException("original must not be null");
```

```
        BigDecimal amount = original.amount();
        if (amount.compareTo(BigDecimal.valueOf(1000)) >= 0) {
            BigDecimal discounted =
amount.multiply(BigDecimal.valueOf(0.90));
            return new Money(discounted);
        }
        return original;
    }
}
```

4.3 Example with dependency: `OrderService`

We'll mock the gateway in tests.

```
package com.example.service;

import com.example.domain.Money;

import java.util.UUID;

public class OrderService {

    public interface PaymentGateway {
        String charge(UUID orderId, Money amount);
    }

    private final PaymentGateway paymentGateway;
    private final DiscountService discountService;

    public OrderService(PaymentGateway paymentGateway,
DiscountService discountService) {
        if (paymentGateway == null) throw new
IllegalArgumentException("paymentGateway must not be null");
        if (discountService == null) throw new
IllegalArgumentException("discountService must not be null");
        this.paymentGateway = paymentGateway;
        this.discountService = discountService;
    }

    public String checkout(UUID orderId, Money subtotal) {
```

```
        if (orderId == null) throw new
IllegalArgumentException("orderId must not be null");
        if (subtotal == null) throw new
IllegalArgumentException("subtotal must not be null");

        Money finalAmount = discountService.applyDiscount(subtotal);

        if (finalAmount.amount().signum() <= 0) {
            throw new IllegalStateException("finalAmount must be >
0");
        }

        return paymentGateway.charge(orderId, finalAmount);
    }
}
```

---

5) Writing JUnit tests: core concepts + examples

Create tests under `src/test/java`.

5.1 Test naming conventions (important)

Good tests read like specs. Typical patterns:

- `methodName_condition_expectedResult`
- `shouldDoX_whenY`
- Use **clear "arrange-act-assert"** structure.

---

6) Your first JUnit 5 test class

`DiscountServiceTest`

```
package com.example.service;

import com.example.domain.Money;
import org.junit.jupiter.api.*;
```

```
import java.math.BigDecimal;

import static org.assertj.core.api.Assertions.*;

class DiscountServiceTest {

    private DiscountService discountService;

    @BeforeEach
    void setUp() {
        discountService = new DiscountService();
    }

    @Test
    void shouldReturnSameAmount_whenBelowDiscountThreshold() {
        Money original = new Money(BigDecimal.valueOf(999));

        Money result = discountService.applyDiscount(original);

        assertThat(result).isEqualTo(original);
        assertThat(result.amount()).isEqualByComparingTo("999");
    }

    @Test
    void shouldApply10PercentDiscount_whenAtOrAboveThreshold() {
        Money original = new Money(BigDecimal.valueOf(1000));

        Money result = discountService.applyDiscount(original);

        assertThat(result.amount()).isEqualByComparingTo("900.0");
    }

    @Test
    void shouldThrowIllegalArgumentException_whenOriginalIsNull() {
        assertThatThrownBy(() ->
discountService.applyDiscount(null))
                .isInstanceOf(IllegalArgumentException.class)
                .hasMessageContaining("original must not be null");
    }
}
```

Concepts covered here

- `@BeforeEach` setup
- Basic `@Test`
- Assertions (AssertJ):
    - equality
    - numeric comparison
    - exception assertions with message checks

---

7) Assertions: all common scenarios (JUnit + AssertJ)

7.1 Basic assertions (JUnit)

JUnit has `org.junit.jupiter.api.Assertions.*`:

- `assertEquals`
- `assertTrue`
- `assertFalse`
- `assertThrows`
- `assertAll`
- `assertTimeout`

Example:

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

@Test
void basicAssertionsExample() {
    assertEquals(4, 2 + 2);
    assertTrue("abc".startsWith("a"));
```

```
    assertThrows(IllegalArgumentException.class, () ->
Integer.parseInt("x"));
}
```

7.2 Grouped assertions (`assertAll`)

Useful when you want multiple checks without stopping at first failure.

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

@Test
void groupedAssertions() {
    String name = "Aarav Sharma";
    assertAll("name checks",
        () -> assertTrue(name.contains("Aarav")),
        () -> assertTrue(name.contains("Sharma")),
        () -> assertEquals(12, name.length())
    );
}
```

---

8) Parameterized tests (cover many cases cleanly)

8.1 `@ValueSource`

```
package com.example.service;

import com.example.domain.Money;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import java.math.BigDecimal;

import static org.assertj.core.api.Assertions.*;

class DiscountServiceParameterizedTest {

    private final DiscountService discountService = new
DiscountService();

    @ParameterizedTest
```

```
    @ValueSource(longs = {0, 1, 999})
    void shouldNotDiscount_whenBelowThreshold(long value) {
        Money original = new Money(BigDecimal.valueOf(value));

        Money result = discountService.applyDiscount(original);

        assertThat(result).isEqualTo(original);
    }
}
```

8.2 `@CsvSource` (multiple inputs and expected)

```
import org.junit.jupiter.params.provider.CsvSource;

@ParameterizedTest
@CsvSource({
    "1000, 900.0",
    "1500, 1350.0",
    "2000, 1800.0"
})
void shouldDiscountCorrectly(long input, String expected) {
    Money original = new Money(BigDecimal.valueOf(input));

    Money result = discountService.applyDiscount(original);

    assertThat(result.amount()).isEqualByComparingTo(expected);
}
```

8.3 `@MethodSource` (complex objects)

```
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

static Stream<org.junit.jupiter.params.provider.Arguments>
discountCases() {
    return Stream.of(
        org.junit.jupiter.params.provider.Arguments.of("1000",
"900.0"),
        org.junit.jupiter.params.provider.Arguments.of("2500",
"2250.0")
```

```
    );
}

@ParameterizedTest
@MethodSource("discountCases")
void discountWithMethodSource(String input, String expected) {
    Money original = new Money(new java.math.BigDecimal(input));
    Money result = discountService.applyDiscount(original);
    assertThat(result.amount()).isEqualByComparingTo(expected);
}
```

---

9) Testing code that uses dependencies (Mockito)

`OrderServiceTest` with mocks

```
package com.example.service;

import com.example.domain.Money;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.*;
import org.mockito.junit.jupiter.MockitoExtension;

import java.math.BigDecimal;
import java.util.UUID;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.ArgumentMatchers.*;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
class OrderServiceTest {

    @Mock
    OrderService.PaymentGateway paymentGateway;

    @Spy
    DiscountService discountService = new DiscountService(); // real
logic, but spy-able if needed
```

```
    @InjectMocks
    OrderService orderService;

    @Test
    void shouldChargeDiscountedAmount_whenEligibleForDiscount() {
        UUID orderId = UUID.randomUUID();
        Money subtotal = new Money(BigDecimal.valueOf(1000));

        when(paymentGateway.charge(eq(orderId), any(Money.class)))
                .thenReturn("TXN-123");

        String txnId = orderService.checkout(orderId, subtotal);

        assertThat(txnId).isEqualTo("TXN-123");

        // Verify exact final amount sent to gateway
        ArgumentCaptor<Money> captor =
ArgumentCaptor.forClass(Money.class);
        verify(paymentGateway).charge(eq(orderId),
captor.capture());

assertThat(captor.getValue().amount()).isEqualByComparingTo("900.0")
;
    }

    @Test
    void shouldThrow_whenFinalAmountIsZeroOrNegative() {
        UUID orderId = UUID.randomUUID();
        Money subtotal = new Money(BigDecimal.ZERO);

        assertThatThrownBy(() -> orderService.checkout(orderId,
subtotal))
                .isInstanceOf(IllegalStateException.class)
                .hasMessageContaining("finalAmount must be > 0");

        verifyNoInteractions(paymentGateway);
    }

    @Test
    void shouldPropagateGatewayResult_whenNoDiscount() {
        UUID orderId = UUID.randomUUID();
```

```
        Money subtotal = new Money(BigDecimal.valueOf(500));

        when(paymentGateway.charge(eq(orderId), any(Money.class)))
                .thenReturn("TXN-999");

        String txnId = orderService.checkout(orderId, subtotal);

        assertThat(txnId).isEqualTo("TXN-999");

        verify(paymentGateway).charge(eq(orderId), argThat(m ->
m.amount().compareTo(BigDecimal.valueOf(500)) == 0));
    }
}
```

Mockito concepts covered

- `@Mock` dependency
- `@Spy` partial real object
- `@InjectMocks` auto-wiring into the service under test
- `when(...).thenReturn(...)` stubbing
- `verify(...)` interaction verification
- `ArgumentCaptor` to inspect what was passed
- `verifyNoInteractions(...)` to ensure nothing external was called

---

10) Test lifecycle annotations (all of them)

- `@BeforeAll` / `@AfterAll` (runs once per class)
- `@BeforeEach` / `@AfterEach` (runs before/after each test)
- Default: JUnit creates a new test instance per test method.
- If you need non-static `@BeforeAll`, use `@TestInstance(PER_CLASS)`.

Example:

```
import org.junit.jupiter.api.*;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class LifecycleDemoTest {

    @BeforeAll
    void beforeAll() { /* runs once */ }

    @BeforeEach
    void beforeEach() { /* runs before each test */ }

    @Test
    void t1() {}

    @Test
    void t2() {}

    @AfterEach
    void afterEach() { /* runs after each test */ }

    @AfterAll
    void afterAll() { /* runs once */ }
}
```

---

11) Nested tests (organize scenarios like a spec)

```
import org.junit.jupiter.api.*;

import static org.assertj.core.api.Assertions.*;

class NestedDemoTest {

    @Nested
    class WhenUserIsEligible {

        @Test
        void shouldApplyBenefit() {
            assertThat(2 + 2).isEqualTo(4);
```

```
        }
    }

    @Nested
    class WhenUserIsNotEligible {

        @Test
        void shouldNotApplyBenefit() {
            assertThat(2 + 2).isNotEqualTo(5);
        }
    }
}
```

Use this when your service has multiple “modes” / “states”.

---

12) Repeated tests + display names

```
import org.junit.jupiter.api.*;

class RepeatedDemoTest {

    @RepeatedTest(value = 3, name = "{displayName} - run
{currentRepetition}/{totalRepetitions}")
    @DisplayName("Retry-like repeated test")
    void repeated() {
        // Use carefully: repeated tests can hide flaky behavior
    }
}
```

---

13) Assumptions (skip tests based on environment)

Use when tests only make sense in certain conditions.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assumptions;

class AssumptionDemoTest {
```

```
    @Test
    void runOnlyOnCi() {

Assumptions.assumeTrue("true".equalsIgnoreCase(System.getenv("CI")))
;
        // test logic
    }
}
```

---

14) Timeouts (prevent tests from hanging)

14.1 JUnit timeout assertion

```
import org.junit.jupiter.api.Test;

import java.time.Duration;

import static org.junit.jupiter.api.Assertions.*;

class TimeoutDemoTest {

    @Test
    void shouldFinishQuickly() {
        assertTimeout(Duration.ofMillis(200), () -> {
            Thread.sleep(50);
        });
    }
}
```

14.2 Preemptive timeout (interrupts running code)

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

@Test
void shouldTimeoutPreemptively() {
    assertTimeoutPreemptively(Duration.ofMillis(100), () -> {
        Thread.sleep(1000);
    });
}
```

Use preemptive carefully if your code doesn’t handle interrupts well.

---

15) Temporary directories / files (no manual cleanup)

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.io.TempDir;

import java.nio.file.*;

import static org.assertj.core.api.Assertions.*;

class TempDirDemoTest {

    @TempDir
    Path tempDir;

    @Test
    void writesFileInTempDir() throws Exception {
        Path file = tempDir.resolve("data.txt");
        Files.writeString(file, "hello");

        assertThat(Files.readString(file)).isEqualTo("hello");
    }
}
```

---

16) Tags: run subsets of tests (fast vs slow)

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

class TagDemoTest {

    @Test
    @Tag("fast")
    void fastTest() {}

    @Test
    @Tag("slow")
    void slowTest() {}
```

```
}
```

Run only fast tests:

```
mvn test -Dgroups=fast
```

(If you want fully correct Maven tag filtering, you typically configure Surefire's includes/excludes or use JUnit Platform properties. I can give the exact Surefire config if you want tag-based builds.)

---

17) Dynamic tests (generated at runtime)

Useful when you generate tests from data, but don't overuse.

```
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

class DynamicTestsDemo {

    @TestFactory
    List<DynamicTest> dynamicTests() {
        return List.of(
            DynamicTest.dynamicTest("2+2=4", () -> assertEquals(4, 2
+ 2)),
            DynamicTest.dynamicTest("3+3=6", () -> assertEquals(6, 3
+ 3))
        );
    }
}
```

---

18) Unit tests vs Integration tests (proper separation)

Unit tests

- Test a class in isolation
- Mock dependencies
- Fast, deterministic

Naming:

- `*Test.java`

Run:

- `mvn test`

Integration tests

- Use real infrastructure (DB, filesystem, HTTP)
- Slower
- Naming:
    - `*IT.java` (common convention)

Run with Failsafe plugin:

- `mvn verify`

Example folder structure:

```
src/test/java/...        (unit tests)
src/test/java/...IT.java (integration tests)
```

Failsafe will run `*IT` by default when configured as earlier.

---

19) Test quality rules (what “good” looks like)

Brutally practical:

- **One reason to fail**: each test should validate one behavior.
- Tests should be:
    - **Readable**
    - **Deterministic**
    - **Fast**
- Avoid:
    - random sleeps
    - dependency on system time/timezone/network
    - over-mocking everything (mock only boundaries)

---

20) Common scenarios checklist (what you should be able to test)

You should confidently cover:

- ✅ happy path
- ✅ boundary values (thresholds, min/max)
- ✅ invalid input validation (nulls, blanks, negatives)
- ✅ exception type + message
- ✅ interaction testing with mocks (`verify`, captor)
- ✅ parameterized “matrix” testing
- ✅ timeouts
- ✅ filesystem with `@TempDir`
- ✅ nested scenario grouping
- ✅ tags for fast/slow suites

- ✅ integration tests using Failsafe

---

21) Run commands you'll actually use

```
mvn -q test
mvn -q test -Dtest=DiscountServiceTest
mvn -q verify
```