Junit Interview Questions

1) What is JUnit and why is it used in Java development?

➔ Junit is a widely-used unit testing framework for Java that allows developers to write and run repeatable automated tests.
➔ It is essential for ensuring code quality and reliability by detecting bugs early in the development process.

2) How do you create a simple test case in Junit ?

➔ "To create a simple test case in JUnit, you need to define a method annotated with @Test. Inside this method, use assertions like assertEquals to verify the expected outcomes."

3) What are annotations in JUnit? Name a few commonly used ones.

➔ Annotations in JUnit are metadata that provide information about the code and are used to configure and control test execution.
➔ Commonly used annotations include @Test, @Before, and @After."

4) How do you run tests in JUnit? Describe the different ways to execute tests.

➔ To run tests in JUnit, you can execute them directly from an IDE like IntelliJ or Eclipse, which provides built-in support for running JUnit tests.
➔ Alternatively, you can use build tools like Maven or Gradle to automate test execution as part of your build process."
➔ There is an option of running tests from the command line using the JUnit platform console launcher.

5) Write a JUnit test case that tests a method throwing an exception.

➔"To write a JUnit test case that tests a method throwing an exception, you can use the @Test(expected = Exception.class) annotation. This ensures that the test will pass only if the specified exception is thrown."

✅ 1. `assertThrows()` (JUnit 5 - most recommended)

Best practice – clean and expressive.

```java
@Test
void testExceptionUsingAssertThrows() {
    Exception exception = assertThrows(
        IllegalArgumentException.class,
        () -> someMethodThatThrows(),
        "Expected IllegalArgumentException"
    );
    assertEquals("Expected message", exception.getMessage());
}
```

✅ 2. `try-catch` block (JUnit 4 & 5)

Old-school but still valid. Allows very custom assertions.

```java
@Test
void testExceptionUsingTryCatch() {
    try {
        someMethodThatThrows();
        fail("Expected exception was not thrown");
    } catch (IllegalArgumentException e) {
        assertEquals("Expected message", e.getMessage());
    }
}
```

✅ 3. `@Test(expected = Exception.class)` (JUnit 4 only)

Concise, but you **can't assert on the message or content** of the exception.

```java
@Test(expected = IllegalArgumentException.class)
public void testExceptionAnnotation() {
    someMethodThatThrows();
}
```

✅ 5. Using AssertJ (fluent API for assertions)

More expressive and chainable, great for readability.

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

@Test
void testWithAssertJ() {
    assertThatThrownBy(() -> someMethodThatThrows())
        .isInstanceOf(IllegalArgumentException.class)
        .hasMessage("Expected message");
}
```

Ques) Explain the concept of test suites in JUnit.

➔ A test suite is like a playlist 🎵… .

➔ Instead of playing one song (test) at a time, you create a playlist of tests and run them all together!

🧪 Why Use a Test Suite?

- ✅ Run multiple test classes together
- ✅ Group related tests (e.g., all login-related tests)
- ✅ Save time & organize better
- ✅ Useful for integration testing

🔧 How to Create a Test Suite in JUnit 5

java Copy Edit

```java
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({
    LoginTest.class,
    RegistrationTest.class,
    ProfileTest.class
})
public class UserFlowTestSuite {
    // This class remains empty — it's just a holder for the annotations above.
}
```

📦 When Should You Use It?

Scenario	Use Suite?
Run all tests across the app	✅
Run tests for a specific feature	✅
Only one test class	❌

Ques) What are Assertions in JUnit?

➔ Assertions are like checkpoints 🛑 in your tests.

➔ They verify that your code is working as expected.

If an assertion fails, the test fails — simple!

✅ Commonly Used JUnit Assertion Methods

🟢 `assertEquals(expected, actual)`

✅ Asserts that two values are equal.

```java
assertEquals(42, mathService.getAnswer());
```

🔴 `assertNotEquals(expected, actual)`

❌ Asserts that two values are not equal.

```java
assertNotEquals(0, user.getAge());
```

🟣 `assertNull(value)`

⚫ Asserts that the value is null.

```java
assertNull(response.getErrorMessage());
```

🔵 `assertTrue(condition)`

🧢 Asserts that a condition is true.

```java
assertTrue(account.isActive());
```

🟠 `assertFalse(condition)`

⛔ Asserts that a condition is false.

```java
assertFalse(user.isBanned());
```

🟢 `assertNotNull(value)`

🟡 Asserts that the value is not null.

```java
assertNotNull(order.getId());
```

🔥 `assertThrows(Exception.class, () -> { ... })`

💥 Asserts that an exception is thrown.

```java
assertThrows(IllegalArgumentException.class, () -> {
    calculator.divide(5, 0);
});
```

Ques) Write a JUnit test case for a method that checks if a string is a palindrome.

```java
public class StringUtils {

    public boolean isPalindrome(String input) {
        if (input == null) return false;
        String clean = input.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();
        return new StringBuilder(clean).reverse().toString().equals(clean);
    }
}
```

```java
@ExtendWith(MockitoExtension.class)
class PalindromeControllerTest {

    @InjectMocks
    PalindromeController controller;

    @Mock
    PalindromeService service;

    @Test
    void testPalindromeTrue() {
        Mockito.when(service.isPalindrome("madam")).thenReturn(true);
        var response = controller.checkPalindrome("madam");
        assertEquals(true, response.getBody());
    }

    @Test
    void testPalindromeFalse() {
        Mockito.when(service.isPalindrome("hello")).thenReturn(false);
        var response = controller.checkPalindrome("hello");
        assertEquals(false, response.getBody());
    }
}
```

Ques) What is the purpose of the @Before and @After annotations?

- ➔ In JUnit, the **@Before** and **@After** annotations are used to define setup and teardown methods that run before and after each test method, respectively.
- ➔ Runs **before each test method**.
- ➔ **@BeforeEach**, Useful for setting up test data, initializing objects, or preparing test environment.
- ➔ **@AfterEach** , Typically used for cleanup, like closing connections or resetting state.

java

```java
@BeforeEach
void setUp() {
    // Code here runs before every @Test
    service = new PalindromeService();
}
```

java Copy Edit

```java
@AfterEach
void tearDown() {
    // Code here runs after every @Test
    service = null;
}
```

Ques) How do you test private methods in JUnit?

Testing private methods directly in JUnit is generally not recommended because:

➔ Private methods are implementation details — they should be tested *indirectly* via public methods.

✅ Preferred Way: Test via Public Methods

If the private method contributes to a public method's behavior, write your test against the public method instead.

java Copy Edit

```java
// PalindromeService has a private "cleanString()" method
// but you test it via the public isPalindrome()

assertTrue(service.isPalindrome("A man, a plan, a canal: Panama"));
```

🛠️ Option 1: Use Reflection (Not Recommended for Regular Use)

java Copy Edit

```java
Method method = PalindromeService.class.getDeclaredMethod("cleanString", String.class);
method.setAccessible(true);
String result = (String) method.invoke(service, "Hello, World!");
assertEquals("helloworld", result);
```

> This breaks encapsulation and should only be used in special cases (e.g., legacy code, static utils, or debugging).

🧪 Option 2: Use a Code Design Refactor

If the logic is complex and really needs direct testing, consider **refactoring the private logic into a package-private or separate helper class.**

java Copy Edit

```java
class StringUtil {
    static String clean(String input) {
        // now test this directly
    }
}
```

🚫 Summary

Approach	Recommended?	Notes
Test via public method	✅ Yes	Cleanest and most maintainable
Use reflection	⚠️ No	Breaks encapsulation, brittle
Refactor to helper class	✅ Yes	Testable and reusable

Ques) Write a JUnit test case that verifies the behavior of a method that interacts with a database.

JUnit Test with H2 In-Memory DB (Recommended Way)

java Copy Edit

```java
@SpringBootTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.ANY)
@Transactional
class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    void testSaveAndFindUser() {
        User user = new User();
        user.setName("Alice");
        userRepository.save(user);

        Optional<User> found = userRepository.findByName("Alice");
        assertTrue(found.isPresent());
        assertEquals("Alice", found.get().getName());
    }
}
```

Ques) How do you integrate JUnit with build tools like Maven or Gradle?

➔Integrating JUnit with build tools like Maven or Gradle is straightforward — both tools have built-in support for JUnit.

✅ With Maven

1. **Add JUnit 5 dependency** in `pom.xml`:

```xml
<dependencies>
    <!-- JUnit 5 (Jupiter) -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2. **Enable the JUnit 5 engine** (if needed):

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.1.2</version> <!-- Make sure it's 2.22.0+ for JUnit 5 -->
        </plugin>
    </plugins>
</build>
```

3. **Run Tests:**

```bash
mvn test
```

Ques) How do we test void methods

✅ 1. Use Mockito to Verify Behavior

➔ If the void method interacts with other components (e.g., calls a repo, logs, sends an email), you verify those interactions using **Mockito.verify().**

✅ Test Case:

java

```java
@ExtendWith(MockitoExtension.class)
class NotificationServiceTest {

    @Mock
    EmailSender emailSender;

    @InjectMocks
    NotificationService notificationService;

    @Test
    void testSendWelcomeEmail() {
        notificationService.sendWelcomeEmail("user@example.com");

        Mockito.verify(emailSender).sendEmail("user@example.com", "Welcome!");
    }
}
```

✅ 3. Use `doNothing()` or `doThrow()` for Mocked Void Methods

For void methods, use `doNothing()`, `doThrow()`:

java Copy Edit

```java
doNothing().when(emailSender).sendEmail(anyString(), anyString());
// or simulate exception
doThrow(new RuntimeException("fail")).when(emailSender).sendEmail(...);
```

✅ 4. **If Method Modifies State, Test the Side Effect**

java Copy Edit

```java
class Counter {
    int count = 0;
    public void increment() {
        count++;
    }
}

@Test
void testIncrement() {
    Counter counter = new Counter();
    counter.increment();
    assertEquals(1, counter.count);
}
```

✅ Summary

Strategy	Use Case
`Mockito.verify()`	Checks that the method did what it's supposed to
`ArgumentCaptor`	Captures arguments passed to the void method
`doThrow()`	Simulate exceptions from void methods
Test side effects	Validate changes in state or object