Method	Purpose
`assertEquals`	Check equality
`assertNotEquals`	Check inequality
`assertTrue` / `False`	Check boolean condition
`assertNull` / `NotNull`	Check for null / non-null
`assertSame` / `NotSame`	Reference equality
`assertArrayEquals`	Compare arrays
`assertIterableEquals`	Compare lists or sets
`assertThrows`	Verify exception is thrown
`assertDoesNotThrow`	Ensure no exception is thrown
`assertAll`	Group multiple assertions
`fail`	Fail test manually

✅ 1. `assertEquals(expected, actual[, message])`

Checks if two values are equal.

```
assertEquals(5, 2 + 3);
assertEquals("abc", "a" + "bc", "Strings should match");
```

✅ 2. `assertNotEquals(unexpected, actual[, message])`

Checks if two values are **not equal**.

```
assertNotEquals(10, 5 + 5);
```

✅ 3. `assertTrue(condition[, message])`

Passes if the condition is `true`.

```
assertTrue(5 > 1, "5 should be greater than 1");
```

✅ 4. `assertFalse(condition[, message])`

Passes if the condition is `false`.

```
assertFalse(2 > 10, "2 should not be greater than 10");
```

✅ 5. `assertNull(object[, message])`

Passes if the object is `null`.

```
String s = null;
assertNull(s);
```

✅ 6. `assertNotNull(object[, message])`

Passes if the object is **not null**.

```
String name = "JUnit";
assertNotNull(name);
```

✅ 7. `assertSame(expected, actual[, message])`

Checks if two object **references** point to the **same object** (`==`).

```
String a = "abc";
String b = a;
assertSame(a, b);
```

✅ 8. `assertNotSame(unexpected, actual[, message])`

Passes if two references **do not** point to the same object.

```
assertNotSame(new String("abc"), new String("abc"));
```

✅ 9. `assertArrayEquals(expectedArray, actualArray[, message])`

Checks if two arrays are equal (same elements, same order).

```
int[] expected = {1, 2, 3};
```

```
int[] actual = {1, 2, 3};
assertArrayEquals(expected, actual);
```

✅ 10. `assertIterableEquals(expectedIterable, actualIterable[, message])`

Checks if two iterables (e.g., lists) are equal.

```
List<String> expected = List.of("A", "B");
List<String> actual = List.of("A", "B");
assertIterableEquals(expected, actual);
```

✅ 11. `assertThrows(expectedException.class, executable)`

Checks that a specific exception is thrown.

```
assertThrows(ArithmeticException.class, () -> {
    int x = 1 / 0;
});
```

✅ 12. `assertDoesNotThrow(executable)`

Asserts that the code block **does not throw** any exceptions.

```
assertDoesNotThrow(() -> {
    int x = 1 + 1;
});
```

✅ 13. `assertAll(...)`

Groups multiple assertions together. All are evaluated, even if some fail.

```
assertAll(
    () -> assertEquals(4, 2 + 2),
    () -> assertTrue("abc".startsWith("a")),
    () -> assertNotNull("hello")
);
```

✅ 14. `fail([message])`

Forces a test to fail. Useful in unreachable code or conditional checks.

```java
fail("Test failed intentionally");
```

We'll keep the User and UserService classes simple and show each Assertions method with **one clear, small example** in its own test class.

✅ Step 1: Base Classes

Create these two classes once. They will be reused by all tests.

📁 User.java

```java
public class User {
    String name;
    String email;
    int age;

    public User(String name, String email, int age) {
        this.name = name;
        this.email = email;
        this.age = age;
    }
}
```

📁 UserService.java

```java
import java.util.*;

public class UserService {
    private List<User> users = new ArrayList<>();

    public User register(String name, String email, int age) {
        if (email == null || !email.contains("@")) {
            throw new IllegalArgumentException("Invalid email");
        }
        User user = new User(name, email, age);
        users.add(user);
        return user;
    }

    public List<User> getAllUsers() {
        return users;
    }

    public boolean isAdult(User user) {
        return user.age >= 18;
    }

    public String[] getUserRoles(User user) {
        return new String[]{"USER", "MEMBER"};
    }

    public void clearAllUsers() {
        users.clear();
    }
}
```

✅ Step 2: Simple JUnit Tests (Split by Assertion Type)

📁 TestEquality.java

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

public class TestEquality {
    UserService service = new UserService();

    @Test
    void testAssertEquals() {
        User user = service.register("Alice", "alice@example.com", 25);
        assertEquals("Alice", user.name); // ✅ Expected: Alice
    }

    @Test
    void testAssertNotEquals() {
        User user = service.register("Bob", "bob@example.com", 30);
        assertNotEquals("Tom", user.name); // ✅ Expected: Not equal
    }
}
```

📁 TestBooleanChecks.java

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

```java
public class TestBooleanChecks {
    UserService service = new UserService();

    @Test
    void testAssertTrue() {
        User user = service.register("Eva", "eva@example.com", 20);
        assertTrue(service.isAdult(user)); // ✅ Expected: true
    }

    @Test
    void testAssertFalse() {
        User user = service.register("Tim", "tim@example.com", 15);
        assertFalse(service.isAdult(user)); // ✅ Expected: false
    }
}
```

📁 TestNullChecks.java

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

public class TestNullChecks {

    @Test
    void testAssertNull() {
        String email = null;
        assertNull(email); // ✅ Expected: null
    }

    @Test
    void testAssertNotNull() {
        String email = "something@example.com";
        assertNotNull(email); // ✅ Expected: not null
    }
}
```

📁 TestObjectIdentity.java

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

public class TestObjectIdentity {

    @Test
    void testAssertSame() {
        User user = new User("Tom", "tom@example.com", 30);
        User ref = user;
        assertSame(user, ref); // ✅ Same object
    }

    @Test
    void testAssertNotSame() {
        User u1 = new User("Tom", "tom@example.com", 30);
        User u2 = new User("Tom", "tom@example.com", 30);
        assertNotSame(u1, u2); // ✅ Different objects
    }
}
```

📁 TestArrayAndList.java

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

public class TestArrayAndList {
    UserService service = new UserService();

    @Test
    void testAssertArrayEquals() {
        User user = service.register("John", "john@example.com", 25);
        String[] expected = {"USER", "MEMBER"};
        assertArrayEquals(expected, service.getUserRoles(user)); // ✅ Same array
content
    }

    @Test
    void testAssertIterableEquals() {
        service.register("U1", "u1@example.com", 20);
        service.register("U2", "u2@example.com", 21);

        List<String> expectedNames = List.of("U1", "U2");
        List<String> actualNames = service.getAllUsers().stream()
            .map(u -> u.name)
            .toList();

        assertIterableEquals(expectedNames, actualNames); // ✅ Same list
    }
}
```

📁 TestExceptions.java

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

```java
public class TestExceptions {
    UserService service = new UserService();

    @Test
    void testAssertThrows() {
        assertThrows(IllegalArgumentException.class, () -> {
            service.register("Bad", "no-at-symbol", 22);
        }); // ✅ Should throw exception
    }

    @Test
    void testAssertDoesNotThrow() {
        assertDoesNotThrow(() -> {
            service.register("Good", "valid@example.com", 22);
        }); // ✅ Should not throw
    }
}
```

📁 TestAssertAllAndFail.java

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

public class TestAssertAllAndFail {
    UserService service = new UserService();

    @Test
    void testAssertAll() {
        User user = service.register("Mark", "mark@example.com", 30);

        assertAll("User checks",
            () -> assertEquals("Mark", user.name),
            () -> assertTrue(user.email.contains("@")),
            () -> assertTrue(user.age > 18)
        ); // ✅ All checks pass
    }

    @Test
    void testFail() {
        // Forcing a failure (example case)
        if (false) {
            fail("This should never happen");
        }
    }
}
```

✅ Summary Table

Assertion	Purpose	Output Example
`assertEquals`	Compares two values	✅ "Alice" == "Alice"
`assertNotEquals`	Checks they are different	✅ "Bob" ≠ "Tom"
`assertTrue`	Condition must be true	✅ 20 > 18
`assertFalse`	Condition must be false	✅ 15 < 18
`assertNull`	Value is null	✅ null
`assertNotNull`	Value is not null	✅ "email@example.com"
`assertSame`	Same reference	✅ Both refer to same object
`assertNotSame`	Different objects	✅ Different instances
`assertArrayEquals`	Arrays have same elements	✅ [A, B] == [A, B]
`assertIterableEquals`	Lists have same elements	✅ List.of("U1", "U2")
`assertThrows`	Code throws expected exception	✅ IllegalArgumentException
`assertDoesNotThrow`	Code runs without exception	✅ No exception
`assertAll`	Group multiple assertions	✅ All pass
`fail()`	Manually fail test	❌ Test fails

Would you like a downloadable Maven/Gradle project with these files pre-configured?

Let's break down `assertThrows()` and `assertDoesNotThrow()` from JUnit 5 in a **very simple way**, using real examples and explaining **why and when** to use them.

✅ `assertThrows()`

📌 Purpose:

To test that a **specific exception is thrown** when a block of code is executed.

> This is useful when you **expect your code to fail** in certain cases (e.g., invalid input, nulls, etc.).
>
> You specifically throw a exception raised input

✅ Example:

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

public class ThrowTest {

    @Test
    void testInvalidEmailThrowsException() {
        // This method is expected to throw an IllegalArgumentException
        assertThrows(IllegalArgumentException.class, () -> {
            new UserService().register("Bob", "invalid-email", 25);
        });
    }
}
```

💡 Output:

✅ Test passes if `IllegalArgumentException` is thrown.
❌ Test fails if no exception or wrong exception is thrown.

✅ `assertDoesNotThrow()`

📌 Purpose:

To test that a block of code **does not throw any exceptions**.

> Useful when you're testing that **valid input** or **safe operations** don't crash.

✅ Example:

java
CopyEdit

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

public class NoThrowTest {

    @Test
    void testValidRegistrationDoesNotThrow() {
        // Should register successfully without any exception
        assertDoesNotThrow(() -> {
            new UserService().register("Alice", "alice@example.com", 22);
        });
    }
}
```

💡 Output:

✅ Test passes if no exceptions are thrown.
❌ Test fails if **any** exception is thrown.

🆚 Summary: `assertThrows` vs `assertDoesNotThrow`

Feature	`assertThrows()`	`assertDoesNotThrow()`
Checks for	Specific exception **must be thrown**	No exception should be thrown
Use when	You expect a method to **fail** safely	You expect the method to run **fine**
Passes if	Correct exception is thrown	No exception is thrown
Fails if	No or wrong exception is thrown	Any exception is thrown

---

Would you like me to combine both into a test suite for practice?

Here's the **syntax** for both `assertThrows()` and `assertDoesNotThrow()` in **JUnit 5**, with clear structure and examples:

✅ `assertThrows()` – Syntax

```
assertThrows(ExpectedExceptionClass.class, () -> {
    // Code that should throw the exception
});
```

🔍 Example:

```
assertThrows(IllegalArgumentException.class, () -> {
    new UserService().register("Tom", "invalid-email", 20);
});
```

✅ `assertDoesNotThrow()` – Syntax

```
assertDoesNotThrow(() -> {
    // Code that should NOT throw any exception
});
```

🔍 Example:

```
assertDoesNotThrow(() -> {
    new UserService().register("Alice", "alice@example.com", 25);
});
```

✅ Optional: Capture returned value (useful in both)

`assertThrows` with returned exception:

```
IllegalArgumentException ex = assertThrows(IllegalArgumentException.class, () -> {
    service.register("Tom", "invalid-email", 20);
});
assertEquals("Invalid email", ex.getMessage());
```

`assertDoesNotThrow` with returned value:

```
User user = assertDoesNotThrow(() -> service.register("Alice", "alice@example.com",
22));
assertEquals("Alice", user.name);
```

Let me know if you want examples using custom exceptions too!

- We define a **custom exception** called `NumberTooLargeException`.
- We create a method that throws this exception if the number is **greater than 10**.
- We then test it using **`assertThrows()`** in a JUnit 5 test case.

✅ Step 1: Create the Custom Exception

java
CopyEdit
```
public class NumberTooLargeException extends RuntimeException {
    public NumberTooLargeException(String message) {
        super(message);
    }
}
```

✅ Step 2: Create a Method That Throws the Exception

java
CopyEdit

```
public class NumberProcessor {

    public void validateNumber(int number) {
        if (number > 10) {
            throw new NumberTooLargeException("Number exceeds the limit of 10");
        }
        System.out.println("Valid number: " + number);
    }
}
```

✅ Step 3: Test Using `assertThrows()`

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

public class NumberProcessorTest {

    NumberProcessor processor = new NumberProcessor();

    @Test
    void testNumberTooLargeExceptionThrown() {
        Exception exception = assertThrows(NumberTooLargeException.class, () -> {
            processor.validateNumber(15); // This should throw an exception
        });

        assertEquals("Number exceeds the limit of 10", exception.getMessage());
    }

    @Test
    void testValidNumberDoesNotThrow() {
        assertDoesNotThrow(() -> {
            processor.validateNumber(8); // Should be fine
        });
    }
}
```

✅ Output:

- First test (`testNumberTooLargeExceptionThrown`) passes if exception is thrown correctly.
- Second test (`testValidNumberDoesNotThrow`) passes because 8 is a valid number.

Would you like to test this with multiple values using `@ParameterizedTest` too?