|  |  |
| --- | --- |
| assertEquals(expected, actual) | Verifies that two values are equal. |

|  |  |
| --- | --- |
| assertTrue(condition) | Verifies that the condition is true. |

|  |  |
| --- | --- |
| assertFalse(condition) | Verifies that the condition is false. |

|  |  |
| --- | --- |
| assertNull(object) | Verifies that the object is null. |

|  |  |
| --- | --- |
| assertNotNull(object) | Verifies that the object is not null. |

|  |  |
| --- | --- |
| assertThrows() | Verifies that a particular exception is thrown. |

public class AssertThrowsExample {

@Test

void exceptionTesting() {

Throwable exception = assertThrows(IllegalArgumentException.class, () -> {

throw new IllegalArgumentException("a message");

});

assertEquals("a message", exception.getMessage());

}

}

@Test

public void testDifferentExceptions() {

// Given

Payment payment = new Payment("12345", 100.0);

doThrow(new IllegalArgumentException("Invalid payment amount")).when(paymentRepository).savePayment(payment);

when(dictMock.getMeaning(anyString())).thenThrow(NullPointerException.class);

// When & Then

IllegalArgumentException exception = assertThrows(IllegalArgumentException.class, () -> {

paymentService.processPayment(payment);

});

assertEquals("Invalid payment amount", exception.getMessage());

}

@ExtendWith(MockitoExtension.class)

public class PaymentServiceTest {

@Mock

private PaymentRepository paymentRepository;

@InjectMocks

private PaymentService paymentService;

@Test

public void testProcessPaymentThrowsException() {

// Given

Payment payment = new Payment("12345", 100.0);

doThrow(new Exception("Payment failed")).when(paymentRepository).savePayment(payment);

// When & Then

Exception exception = assertThrows(Exception.class, () -> {

paymentService.processPayment(payment);

});

assertEquals("Payment failed", exception.getMessage());

}

@Test

public void testGetPaymentThrowsException() {

// Given

String transactionId = "12345";

doThrow(new Exception("Payment not found")).when(paymentRepository).findPaymentByTransactionId(transactionId);

// When & Then

Exception exception = assertThrows(Exception.class, () -> {

paymentService.getPayment(transactionId);

});

assertEquals("Payment not found", exception.getMessage());

}

}

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>4.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.9.2</version>

<scope>test</scope>

</dependency>

@Test

public void assumeTrueWithNoMessage() {

assumeTrue("DEV".equals(System.getProperty("ENV")));

System.out.println("Assumption passed !!!");

assertEquals(3, 2 + 1);

}

@ExtendWith(MockitoExtension.class)

public class LibraryServiceTest {

@Mock

private BookRepository bookRepository;

@InjectMocks

private LibraryService libraryService;

@Test

public void testGetBookDetails() {

// Given

String title = "Mockito in Action";

Book book = new Book(title, "Sanjay Kumar");

doReturn(book).when(bookRepository).findBookByTitle(title);

// When

Book result = libraryService.getBookDetails(title);

// Then

assertNotNull(result);

assertEquals(title, result.getTitle());

assertEquals("Sanjay Kumar", result.getAuthor());

}

The verify() method in Mockito is used to check if certain methods on mock objects were called with specific arguments.

@Test

public void testAddBook() {

// Given

String title = "Effective Java";

String author = "Joshua Bloch";

// When

libraryService.addBook(title, author);

// Then

Book book = new Book(title, author);

verify(bookRepository).saveBook(book);

}

}

@ExtendWith(MockitoExtension.class)

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

public void testRegisterUser() {

// Given

String name = "John Doe";

String email = "john.doe@example.com";

// When

userService.registerUser(name, email);

// Then

verify(userRepository).saveUser(new User(name, email));

}

@Test

public void testGetUserByEmail() {

// Given

String email = "jane.doe@example.com";

User user = new User("Jane Doe", email);

when(userRepository.findUserByEmail(email)).thenReturn(user);

// When

User result = userService.getUserByEmail(email);

// Then

assertNotNull(result);

assertEquals("Jane Doe", result.getName());

assertEquals(email, result.getEmail());

verify(userRepository).findUserByEmail(email);

}

}

The @Captor annotation is used to create an argument captor, which captures argument values passed to mocked methods.

@ExtendWith(MockitoExtension.class)

public class CaptorExampleTest {

@Mock

private List<String> mockList;

@Captor

private ArgumentCaptor<String> captor;

@Test

public void testCaptor() {

mockList.add("test");

verify(mockList).add(captor.capture());

assertEquals("test", captor.getValue());

}

}

**@Spy**

The @Spy annotation is used to create a spy of a real object. Spies allow partial mocking, where you can mock some methods while using real implementations for others.

**Example**

In this example, we will create a spy for a User class and mock its getName method.

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Spy;

import org.mockito.junit.jupiter.MockitoExtension;

import org.junit.jupiter.api.extension.ExtendWith;

class User {

public String getName() {

return "Real Name";

}

}

@ExtendWith(MockitoExtension.class)

public class SpyExampleTest {

@Spy

private User user;

@Test

public void testSpy() {

when(user.getName()).thenReturn("Mocked Name");

String name = user.getName();

assertEquals("Mocked Name", name);

}

}