



Guía de Estudio: Mockito

Framework para Testing en Java

Una guía completa para dominar el framework de pruebas unitarias más popular de Java

Conceptos, anotaciones, configuración y mejores prácticas



Preparado por: Equipo de Desarrollo

Version 5.x



Made with Genspark

Índice de Contenidos

- 1** Introducción
- 2** ¿Qué es Mockito?
- 3** ¿Por qué usar Mockito?
- 4** Instalación y configuración
- 5** Conceptos fundamentales
- 6** Anotaciones principales
- 7** Creación de mocks
- 8** Stubbing
- 9** Verificación de interacciones
- 10** Inyección de dependencias

- 11** Spies vs Mocks
- 12** Argument Matchers
- 13** Captura de argumentos
- 14** Mocking de métodos estáticos
- 15** Mejores prácticas
- 16** Ejemplos prácticos
- 17** Mockito vs otros frameworks
- 18** Recursos adicionales
- 19** Ejercicios prácticos
- 20** Conclusiones

 Haz clic en cada sección para acceder directamente al contenido

⚠ ¿Qué es Mockito?

Definición

Mockito es un popular framework de pruebas de código abierto para Java que permite la creación de objetos simulados (mocks) en pruebas automatizadas.

Facilita el aislamiento de componentes para realizar pruebas unitarias efectivas mediante la simulación del comportamiento de dependencias externas.

Propósito Principal

Permitir probar unidades de código de forma **aislada**, sin depender del comportamiento real de otros componentes.

Características Principales

- ✓ API Clara y Simple
Sintaxis intuitiva que facilita la escritura de pruebas
- ⚙️ Configuración de Comportamiento
Define respuestas específicas para llamadas a métodos
- 🔍 Verificación de Interacciones
Comprueba si ciertos métodos fueron llamados correctamente
- ⊕ Flexibilidad
Compatible con JUnit y otros frameworks de testing

“ Mockito es a las pruebas unitarias lo que un doble de riesgo es a un actor: permite probar sin riesgos ”

mock(Database.class)



Objeto simulado listo para testing



Made with Genspark

💡 ¿Por qué usar Mockito?

Beneficios Principales

Aislamiento de Componentes

Permite probar unidades de código de forma independiente, sin dependencias externas

Desarrollo Más Rápido

Acelera el ciclo de pruebas al eliminar configuraciones complejas y dependencias externas

Detección Temprana de Errores

Identifica problemas en la fase de desarrollo antes de llegar a producción

Ventajas Técnicas

API Clara y Fluida

Sintaxis intuitiva y expresiva que facilita la escritura y lectura de pruebas

Verificación Precisa

Comprueba exactamente cómo y cuántas veces se llamaron los métodos simulados

Integración Perfecta

Compatible con JUnit, TestNG y principales frameworks de testing en Java

Impacto en el Desarrollo



Sin Mockito

- Pruebas con dependencias reales y complejas
- Mayor tiempo en configuración de entornos
- Pruebas lentas y frágiles ante cambios



Con Mockito

- Pruebas aisladas y enfocadas en comportamiento
- Configuración sencilla y rápida
- Pruebas rápidas y mantenibles



⬇️ Instalación y Configuración



Maven

Agrega las siguientes dependencias a tu archivo pom.xml:

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-core -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>
```

Nota: La dependencia mockito-junit-jupiter es necesaria para la integración con JUnit 5.



Gradle

Agrega las siguientes dependencias a tu archivo build.gradle:

```
// Para proyectos Java
dependencies {
    testImplementation 'org.mockito:mockito-core:5.5.0'
    testImplementation 'org.mockito:mockito-junit-jupiter:5.5.0'
}

// Para proyectos Android
dependencies {
    testImplementation 'org.mockito:mockito-core:5.5.0'
    androidTestImplementation 'org.mockito:mockito-android:5.5.0'
}
```



Consejo: Siempre usa la versión más reciente compatible con tu proyecto.

Pasos para empezar con Mockito:

- 1 Agregar dependencias al proyecto
- 3 Crear mocks de objetos usando mock()
- 2 Importar clases de Mockito en tus tests
- 4 Configurar comportamiento con when()



Conceptos Fundamentales de Mockito



Mock Objects

Definición: Objetos simulados que imitan el comportamiento de objetos reales de manera controlada.

Propósito: Reemplazar dependencias externas para aislar la unidad que se está probando.

Características:

- No mantienen comportamiento real de la clase original
- Devuelven valores predeterminados (null, 0, false)
- Permiten configurar respuestas específicas
- Permiten verificar interacciones

```
List<String> mockedList =  
mock(List.class);
```



Stubs

Definición: Mocks con comportamiento predefinido para responder de manera específica a ciertas llamadas.

Propósito: Proporcionar valores de retorno controlados para simular escenarios específicos.

Características:

- Se crean utilizando la sintaxis when/thenReturn
- No verifican interacciones por sí mismos
- Pueden configurarse para lanzar excepciones
- Útiles para simular respuestas de servicios externos

```
when(mockedList.get(0)).thenReturn("first");
```



Spies

Definición: Wrappers de objetos reales que registran las interacciones sin alterar su comportamiento original.

Propósito: Monitorear llamadas a métodos mientras se mantiene la funcionalidad real.

Características:

- Llaman a los métodos reales por defecto
- Permiten override selectivo de métodos
- Útiles cuando se necesita funcionalidad parcial
- Permiten verificar interacciones

```
List<String> spy = spy(new  
ArrayList<>());
```

 ¿Cuándo usar cada concepto?

Mocks: Cuando necesitas aislar completamente la unidad de prueba.

Stubs: Cuando necesitas simular respuestas específicas de dependencias.

Spies: Cuando necesitas mantener el comportamiento real pero también verificar interacciones.

Anotaciones Principales en Mockito

@Mock Creación de Mocks

Crea un objeto simulado de la clase o interfaz especificada. Es la anotación más utilizada en Mockito.

```
@Mock  
List<String> mockedList;
```

Requiere inicialización con

 MockitoAnnotations.openMocks(this) o
 @ExtendWith(MockitoExtension.class)

@InjectMocks Inyección de Dependencias

Inyecta automáticamente los mocks creados en el objeto anotado con @InjectMocks.

```
@Mock UserService userService;  
@InjectMocks UserController controller;
```

 Mockito intenta inyectar los mocks mediante: constructor, setter o inyección directa de campos, en ese orden.

@Spy Espías Parciales

Crea un espía parcial que ejecuta los métodos reales a menos que se indique lo contrario.

```
@Spy  
ArrayList<String> spyList = new ArrayList<>();
```

 Con spies, usar doReturn() en lugar de when() para evitar llamadas a métodos reales

@Captor Captura de Argumentos

Permite capturar argumentos pasados a métodos para verificación detallada.

```
@Captor  
ArgumentCaptor<List<String>> listCaptor;
```

Uso típico:

```
verify(mockedList).addAll(listCaptor.capture());  
List<String> capturedArgument = listCaptor.getValue();
```

Ejemplo Completo de Uso

```
import static org.mockito.Mockito.*;  
import org.mockito.Mock;  
import org.mockito.InjectMocks;  
import org.mockito.Spy;  
import org.mockito.Captor;  
import org.mockito.junit.jupiter.MockitoExtension;  
  
@ExtendWith(MockitoExtension.class)  
class UserServiceTest {  
    @Mock UserRepository repository;  
    @Spy EmailValidator validator = new EmailValidator();  
    @Captor ArgumentCaptor<User> userCaptor;  
    @InjectMocks UserService service;  
}
```

Creación de Mocks Básicos

</> Método mock()

Crea mocks programáticamente usando el método estático `mock()`:

```
// Importaciones necesarias
import static org.mockito.Mockito.mock;

// Crear un mock de una lista
List<String> mockedList = mock(List.class);

// Usar el mock (sin comportamiento definido)
mockedList.add("elemento"); // no hace nada
String valor = mockedList.get(0); // retorna null

// A partir de Mockito 4.10.0
List<String> mockedList = mock(); // inferencia de tipo
```

Nota: Los mocks retornan valores por defecto (null, 0, false) si no se configura su comportamiento.

@ Usando @Mock

Crea mocks usando anotaciones para un código más limpio:

```
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class MiTest {

    @Mock
    List<String> mockedList;

    @Test
    void testMockList() {
        // El mock ya está creado y listo para usar
        mockedList.add("elemento");
        verify(mockedList).add("elemento");
    }
}
```

Consejo: Con JUnit 4, necesitarás inicializar los mocks con `MockitoAnnotations.initMocks(this)` en el método `@Before`.

Consideraciones importantes:

- 1 Los mocks no conservan ningún estado interno
- 2 El enfoque con anotaciones requiere extensión/regla de JUnit
- 3 Se pueden crear mocks de interfaces y clases concretas
- 4 Para clases abstractas o finales, usa `spy()` o Mockito 4+

Stubbing - Configuración de Comportamiento

when().thenReturn()

Configura el valor de retorno cuando se llama a un método en un mock:

```
// Crear mock
List<String> mockedList = mock(List.class);

// Definir valor de retorno
when(mockedList.get(0)).thenReturn("first");
when(mockedList.size()).thenReturn(1);

// Estos métodos devolverán los valores configurados
System.out.println(mockedList.get(0)); // Imprime: "first"
System.out.println(mockedList.size()); // Imprime: 1

// Valor no configurado = null
System.out.println(mockedList.get(1)); // Imprime: null
```

Truco: Puedes encadenar valores de retorno para llamadas sucesivas:

```
when(mock.method()).thenReturn("A").thenReturn("B");
// Equivalente a:
when(mock.method()).thenReturn("A", "B");
```

when().thenThrow()

Configura excepciones a lanzar cuando se llama a un método en un mock:

```
// Crear mock
Map<String, String> mockedMap = mock(Map.class);

// Configurar para lanzar excepciones
when(mockedMap.get("clave")).thenThrow(new
RuntimeException("error simulado"));

try {
    mockedMap.get("clave"); // Lanzará RuntimeException
} catch (RuntimeException e) {
    System.out.println(e.getMessage()); // "error simulado"
}

// Ejemplo con clase de excepción
when(mockedMap.put(anyString(),
null)).thenThrow(NullPointerException.class);
```

Importante: Si el método devuelve void, debes usar doThrow() en lugar de when().thenThrow():

```
doThrow(new IOException()).when(mock).closeConnection();
```

Patrones comunes de stubbing:

1 Valores múltiples (first call, second call):

```
when(mock.getValue())
    .thenReturn("uno")
    .thenReturn("dos")
    .thenThrow(RuntimeException.class);
```

2 Coincidencia flexible con matchers:

```
when(mock.process(anyString(), eq(42)))
    .thenReturn(true);
```

✓ Verificación de Interacciones

🔍 Método verify() básico

Mockito permite verificar que determinados métodos han sido llamados en los mocks:

```
// Creación del mock
List<String> mockedList = mock(List.class);

// Uso del mock
mockedList.add("elemento");
mockedList.clear();

// Verificación
verify(mockedList).add("elemento");
verify(mockedList).clear();
```

💡 Importante: Por defecto, verify() verifica que el método fue llamado exactamente una vez.

☰ Variantes de verify()

Mockito ofrece varias opciones para verificar el número de llamadas:

```
// Verificar número exacto de llamadas
verify(mockedList, times(2)).get(0);

// Verificar que nunca se llamó
verify(mockedList, never()).isEmpty();

// Verificar al menos n llamadas
verify(mockedList, atLeast(1)).size();

// Verificar como máximo n llamadas
verify(mockedList, atMost(3)).clear();
```

💡 Consejo: Utiliza verifyNoMoreInteractions(mock) para asegurar que no hay más interacciones no verificadas.

Verificaciones avanzadas:

```
// Verificar orden de llamadas
InOrder inOrder = inOrder(mockedList);
inOrder.verify(mockedList).add("primero");
inOrder.verify(mockedList).add("segundo");
```

```
// Verificar con matchers de argumentos
verify(mockedList).add(anyString());
verify(mockedList).set(eq(0), any(String.class));
```

⚠️ Advertencia: Si ejecutas verify para un método que no ha sido llamado, Mockito lanzará VerificationError. Utiliza never() si esperas que no sea llamado.



Inyección de Dependencias con `@InjectMocks`

¿Qué es `@InjectMocks`?

La anotación `@InjectMocks` crea una instancia real del objeto bajo prueba e inyecta automáticamente los mocks creados con `@Mock` o `@Spy`.

Permite probar clases con dependencias de manera más simple, reduciendo el código repetitivo para la configuración de pruebas.

Tipos de Inyección

Mockito intenta inyectar mocks en este orden:

Constructor

Usa el constructor con más parámetros que coincidan

Setter

Busca métodos setter para cada dependencia

Campo

Inyecta directamente en los campos de la clase

Ejemplo de Uso

```
@ExtendWith(MockitoExtension.class)
class ArticleManagerTest {
    @Mock
    ArticleDatabase database;
    @Mock
    UserService userService;
    @InjectMocks
    ArticleManager manager;
    @Test
    void testArticleManager() {
        // Los mocks ya están inyectados
        // en manager automáticamente
        manager.addArticle("Título", "Contenido");
        verify(database).saveArticle(any(Article.class));
    }
}
```

Ventajas

-  Código más limpio
Evita escribir manualmente código de inicialización
-  Mantenibilidad
Actualización automática al cambiar dependencias
-  Menos errores
Reduce posibilidad de olvidar inyectar dependencias

Crear mocks con `@Mock`

```
@Mock UserService userService;
```

Anotar clase con `@InjectMocks`

```
@InjectMocks ArticleManager manager;
```

Iniciar mocks

```
MockitoAnnotations.openMocks(this);
```

¡Listo para usar!

```
manager.addArticle();
```

➡ Spies vs Mocks



MOCK

Objeto simulado completo con comportamiento predefinido

"Simula toda la clase"

VS



SPY

Objeto real con capacidad de observación y modificación parcial

"Observa comportamiento real"

Comparación de características

Característica	Mock	Spy
Comportamiento	Por defecto, no hace nada	Llama al método real
Creación	mock(Class) o @Mock	spy(Object) o @Spy
Stubbing	when().thenReturn()	doReturn().when()

Ejemplo de Mock

```
@Test
void testWithMock() {
    // Crea un mock completo
    List<String> mockedList = mock(List.class);

    // Define comportamiento
    when(mockedList.get(0)).thenReturn("Primero");

    // El método no configurado retorna null
    assertNull(mockedList.get(1));
}
```

Ejemplo de Spy

```
@Test
void testWithSpy() {
    // Crea un spy sobre objeto real
    List<String> realList = new ArrayList<>();
    List<String> spyList = spy(realList);

    // Modifica solo un método específico
    doReturn("Personalizado").when(spyList).get(0);

    // Los demás métodos funcionan normalmente
    spyList.add("Elemento real");
}
```

¿Cuándo usar cada uno?

✓ **Mock:** Cuando necesitas aislar completamente tu unidad de prueba de sus dependencias.

✓ **Spy:** Cuando quieres mantener el comportamiento real de la mayoría de los métodos, modificando solo algunos.



Made with Genspark

Argument Matchers en Mockito

</> Matchers Básicos

Los matchers permiten flexibilidad al hacer stubbing o verificación sin usar valores exactos:

```
// Cualquier String  
when(mockedList.add(anyString())).thenReturn(true);  
  
// Cualquier Integer  
when(mockedList.get(anyInt())).thenReturn("elemento");  
  
// Cualquier tipo  
when(mockedList.contains(any())).thenReturn(true);  
  
// Valor específico  
when(mockedList.contains(eq("test"))).thenReturn(true);
```

Importante: Cuando mezclas matchers con valores exactos en el mismo método, debes usar eq() para todos los valores exactos.

Matchers Avanzados

Matchers para casos más específicos:

```
// Comparaciones numéricas  
verify(mockedList).addAll(argThat(list -> list.size() > 2));  
  
// Matchers de Strings  
when(mockedList.get(intThat(i -> i < 10)))  
.thenReturn("elemento válido");  
  
// Combinando matchers  
when(calculator.add(anyInt(), gt(10)))  
.thenReturn(100);
```

Tipos de Matchers

anyInt(), anyDouble() **TIPO**

eq(valor) **VALOR**

startsWith("abc") **STRING**

argThat(predicate) **CUSTOM**

Ejemplo de uso completo:

```
@Test  
void testServiceWithMatchers() {  
    // Configurando el comportamiento del mock  
    when(userRepository.findById(anyInt())).thenReturn(new User("John"));  
    when(userRepository.save(argThat(user -> user.getName().equals("John")))).thenReturn(true);  
  
    // Llamando al método bajo prueba  
    boolean result = userService.updateUser(5, "John");  
  
    // Verificando interacciones con matchers  
    verify(userRepository).findById(eq(5));  
    verify(userRepository).save(any(User.class));  
}
```



Q Captura de Argumentos con @Captor

ArgumentCaptor

Permite capturar los argumentos pasados a los métodos mockeados durante la verificación:

```
import org.mockito.ArgumentCaptor;
import static org.mockito.Mockito.verify;

// Crear un captor
ArgumentCaptor<List<String>> captor =
ArgumentCaptor.forClass(List.class);

// Ejecutar el código bajo prueba
mockedList.addAll(Arrays.asList("one", "two"));

// Verificar y capturar el argumento
verify(mockedList).addAll(captor.capture());

// Obtener el valor capturado
List<String> capturedArgument = captor.getValue();
assertThat(capturedArgument, hasItem("one"));
```

Importante: Usar ArgumentCaptor te permite verificar no solo que se llamó un método, sino también qué argumentos exactos se pasaron.

@Captor

La anotación @Captor simplifica la creación de captores de argumentos:

```
import org.mockito.Captor;
import org.mockito.ArgumentCaptor;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class MiClaseTest {

    @Mock
    List<String> mockedList;

    @Captor
    ArgumentCaptor<List<String>> argumentCaptor;

    @Test
    void debeCapturarArgumentos() {
        // Usar directamente sin inicializar
        mockedList.addAll(Arrays.asList("uno", "dos"));

        verify(mockedList).addAll(argumentCaptor.capture());
        assertEquals(2, argumentCaptor.getValue().size());
    }
}
```

Consejo: Usa getAllValues() para obtener todos los valores capturados cuando el método se llama múltiples veces.

Casos de uso comunes para captura de argumentos:

- 1 Verificar contenido específico de objetos complejos
- 2 Capturar argumentos para análisis posteriores
- 3 Verificar transformaciones de datos en servicios
- 4 Examinar objetos calculados o generados dinámicamente

⚡ Mocking de Métodos Estáticos y Clases Finales

Métodos Estáticos

A partir de Mockito 3.4.0, es posible hacer mock de métodos estáticos utilizando **MockedStatic**.

```
// Clase con método estático
public class Utility {
    public static String getConnection() {
        return "Database connection";
    }
}

// Test con MockedStatic
@Test
void testStaticMethod() {
    try (MockedStatic<Utility> utilities =
        mockStatic(Utility.class)) {

        utilities.when(Utility::getConnection)
            .thenReturn("Mock connection");

        assertEquals("Mock connection",
                    Utility.getConnection());
    }
}
```

 Es necesario usar **try-with-resources** para liberar recursos automáticamente.

Clases Finales

Desde Mockito 2.1.0, se pueden hacer mocks de clases finales sin configuración adicional.

```
// Clase final
final class FinalClass {
    public String getData() {
        return "Real data";
    }
}

// Test con mock de clase final
@Test
void testFinalClass() {
    FinalClass finalMock = mock(FinalClass.class);

    when(finalMock.getData())
        .thenReturn("Mocked data");

    assertEquals("Mocked data",
                finalMock.getData());
}
```

 Funciona mediante la creación de submockers usando Byte Buddy y un archivo de configuración especial.

Limitaciones y Consideraciones

Limitaciones

- No todos los métodos estáticos pueden ser mockeados (métodos nativos)
- Mayor consumo de memoria al mockear estáticos
- No funciona con clases en Java 8 compiladas con JDK anterior a 1.8.0_102

Mejores Prácticas

- Evitar mockear estáticos cuando sea posible (dificulta el testing)
- Usar mockStatic sólo dentro de bloques try-with-resources
- Considerar la inyección de dependencias en lugar de métodos estáticos



PowerMock era necesario antes para métodos estáticos



Mockito 3.4.0+ lo soporta nativamente

✓ Mejores Prácticas y Patrones en Mockito

✓ Qué Hacer (Do's)

- ✓ **Mock solo interfaces externas** - Enfócate en mockear solo las dependencias externas, no el objeto bajo prueba
- ✓ **Mantén los tests legibles** - Usa métodos auxiliares para reducir duplicación y mejorar mantenibilidad
- ✓ **Verifica solo interacciones relevantes** - Enfócate en lo que es importante para el comportamiento que estás probando
- ✓ **Usa @ExtendWith(MockitoExtension.class)** - Con JUnit 5 para inicializar mocks automáticamente

✗ Qué Evitar (Don'ts)

- ✗ **No mockees todo** - El exceso de mocks dificulta el mantenimiento y reduce la confianza en las pruebas
- ✗ **No mockees tipos que no controlas** - Evita mockear clases de bibliotecas de terceros o del JDK
- ✗ **No mockees objetos de valor** - Los objetos inmutables como String, Integer, etc. no deben ser mockeados
- ✗ **Evita stubbing excesivo** - Stubbear demasiados métodos puede significar que tu diseño tiene problemas

Patrones Comunes Recomendados

- 💡 **Patrón AAA (Arrange-Act-Assert)**
Estructura tus tests en 3 secciones: preparación (mocks), acción (llamada al método) y verificación
- 👥 **Test Double por Interfaz**
Diseña tus clases para depender de interfaces, facilitando así el mockeo
- 📝 **Verificación Comportamental**
Verifica cómo interactúa tu código con las dependencias, no solo el resultado final

Consejos para Pruebas Efectivas

- 💡 Utiliza **BDDMockito** para una sintaxis más legible (given/when/then)
- 🛠️ Aprovecha **ArgumentCaptor** para pruebas más flexibles de argumentos complejos
- ⌚ Usa **@Spy** cuando necesites comportamiento real parcial en tus objetos de prueba
- 📝 Evita **setup() compartido excesivo** - cada test debe ser independiente y claro

“ Mockea lo menos posible, prueba lo más posible, y mantén tus tests tan claros como tu código de producción ”

✓ Ejemplos Prácticos de Casos de Uso

Testing de Capa de Servicio Caso común

Probar un servicio que depende de un repositorio para acceder a datos:

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    void test GetUserById() {
        // Arrange
        User expectedUser = new User("1", "John");
        when(userRepository.findById("1")).thenReturn(Optional.of(expectedUser));

        // Act
        User result = userService.getUserById("1");

        // Assert
        assertEquals(expectedUser, result);
        verify(userRepository).findById("1");
    }
}
```

Patrón: Mockear dependencias externas para aislar la unidad bajo prueba.

Testing de Cliente API API Externa

Probar un cliente que consume una API externa:

```
@Test
public void testFetchWeatherData() throws Exception {
    // Mock de RestTemplate
    RestTemplate restTemplateMock = mock(RestTemplate.class);

    // Configuración del servicio con el mock
    WeatherService weatherService = new WeatherService(restTemplateMock);

    // Respuesta esperada
    WeatherData expectedData = new WeatherData("Madrid", 25.5);

    // Configuración del comportamiento del mock
    when(restTemplateMock.getForObject(
        eq("https://api.weather.com/current?city=Madrid"),
        eq(WeatherData.class)
    )).thenReturn(expectedData);

    // Llamada al método bajo prueba
    WeatherData result = weatherService.getWeatherForCity("Madrid");

    // Verificaciones
    assertEquals(expectedData.getCity(), result.getCity());
    assertEquals(expectedData.getTemperature(), result.getTemperature());
}
```

Beneficio: Evita llamadas reales a APIs externas durante las pruebas, lo que las hace más rápidas y confiables.

Testing Asíncrono con Callbacks Avanzado

```
@Test
void testAsyncProcessWithCallback() {
    // 1. Crear los mocks
    AsyncService asyncService = mock(AsyncService.class);
    Callback callback = mock(Callback.class);

    // 2. Configurar el comportamiento con ArgumentCaptor
    ArgumentCaptor<ResultCallback> callbackCaptor = ArgumentCaptor.forClass(ResultCallback.class);

    // 3. Llamada al método bajo prueba
    ProcessorService processor = new ProcessorService(asyncService);
    processor.processDataAsync("input", callback);

    // 4. Verificar que se llamó al servicio asíncrono
    verify(asyncService).processAsync(eq("input"), callbackCaptor.capture());

    // 5. Simular respuesta asíncrona exitosa
    callbackCaptor.getValue().onSuccess("processed data");

    // 6. Verificar que se llamó al callback original con el resultado
    verify(callback).onComplete("processed data");
}
```

⚖️ Mockito vs Otros Frameworks de Testing



Mockito

Framework más popular para mocking en Java

Recomendado



EasyMock

Uno de los primeros frameworks de mocking para Java

Clásico



JMockit

Framework potente con capacidades avanzadas

Avanzado



PowerMock

Extensión para Mockito/EasyMock con capacidades adicionales

Especializado

Comparación de características

Característica	Mockito	EasyMock	JMockit	PowerMock
Facilidad de uso	★★★	★★	★★	★
API Limpia	✓	✓	-	-
Métodos estáticos	✓ (v3+)	✗	✓	✓
Clases finales	✓ (v2+)	✗	✓	✓
Mantenimiento	✓ Alto	- Medio	- Bajo	- Medio

¿Cuándo usar cada framework?

🧪 **Mockito:** Para la mayoría de casos de uso. Especialmente cuando necesitas una API limpia y fácil de aprender.

🧩 **EasyMock:** Si ya estás familiarizado con su API y prefieres su enfoque de "record-and-replay".

📝 **JMockit:** Para escenarios complejos donde necesitas capacidades avanzadas de mocking integradas.

⚡ **PowerMock:** Cuando necesitas mockear métodos estáticos o privados en código legacy que no puedes refactorizar.

Recursos Adicionales y Referencias

Documentación Oficial

-  Sitio Web Oficial
site.mockito.org - Documentación actualizada y guías
-  GitHub de Mockito
github.com/mockito/mockito - Código fuente y ejemplos
-  Javadoc API
javadoc.io/doc/org.mockito/mockito-core - Referencia técnica completa

Tutoriales y Guías

-  Baeldung
Guías detalladas sobre características de Mockito
-  Vogella Tutorials
Tutoriales paso a paso para principiantes y avanzados
-  JavaCodeGeeks
Ejemplos prácticos y casos de uso reales

Videos y Cursos

-  YouTube
Canales recomendados: "Un Programador Nace", "Programando en Java"
-  Udemy
Cursos completos de testing con Mockito y JUnit
-  Pluralsight
Cursos avanzados con ejercicios prácticos

Comunidad y Soporte

-  Stack Overflow
Preguntas y respuestas de la comunidad con tag [mockito]
-  Google Groups
Grupo oficial "mockito" para consultas y discusión
-  Comunidades Java
Canales de Discord y Slack para desarrolladores Java

Libro Recomendado



"Practical Unit Testing with JUnit and Mockito" - Tomek Kaczanowski
Referencia completa para implementación de pruebas unitarias efectivas



Made with Genspark

Ejercicios y Laboratorios Prácticos

Ejercicios Básicos

1. Crear y configurar mocks

Crea un mock de una interfaz List y configura su comportamiento para que retorne valores específicos para get(0) y size().

2. Verificación de interacciones

Simula una clase de servicio que llama a un repositorio y verifica que los métodos correctos fueron invocados.

Laboratorio: Sistema de Biblioteca

Implementa un sistema con las siguientes clases:

- Book (con título, autor, ISBN)
- LibraryRepository (interfaz para acceso a datos)
- LibraryService (lógica de negocio)

Escribe pruebas unitarias completas para LibraryService utilizando mocks para aislar las dependencias.

Ejercicios Avanzados

3. Captura de argumentos

Utiliza ArgumentCaptor para verificar que se pasaron los argumentos correctos en una llamada a método.

```
@Captor  
ArgumentCaptor<User> userCaptor;  
  
verify(userService).createUser(userCaptor.capture());  
assertEquals("nombre", userCaptor.getValue().getName());
```

4. Spies vs Mocks

Implementa la misma prueba usando un mock y un spy, y compara las diferencias en comportamiento.

Desafío: API REST

Crea un conjunto de pruebas para un controlador REST que:

1. Use @InjectMocks para el controlador
2. Mock para los servicios
3. Verifique los códigos de estado HTTP
4. Capture y valide los objetos DTO pasados entre capas

Consejo: Utiliza MockMvc en combinación con Mockito para pruebas completas de API.

Recursos para los ejercicios

 [Repository de código de ejemplo](#)

 [Guía de soluciones](#)

 [Tutoriales en video](#)

🚩 Resumen y Conclusiones

✓ Lo que has aprendido

-  Fundamentos de Mockito y su propósito en el testing
-  Configuración de mocks y stubs para simular dependencias
-  Verificación de interacciones con objetos simulados
-  Inyección de dependencias y manejo de relaciones
-  Patrones y mejores prácticas para pruebas efectivas

👉 Próximos pasos

-  Aplicar Mockito en un proyecto personal o laboral
-  Implementar TDD utilizando Mockito como herramienta clave
-  Explorar integración con otros frameworks (Spring, JUnit 5)
-  Compartir conocimientos y prácticas con tu equipo

“Las pruebas unitarias con Mockito no solo mejoran la calidad del software, sino que también hacen el proceso de desarrollo más eficiente y predecible.”



¡Felicidades por completar esta guía de Mockito!

Ahora tienes las herramientas para crear pruebas unitarias robustas y mantener tu código de alta calidad.