Basic Features of Mockito

Mike Nolan
mnolanjr@gmail.com

pluralsight
hardcore developer training

Module Overview

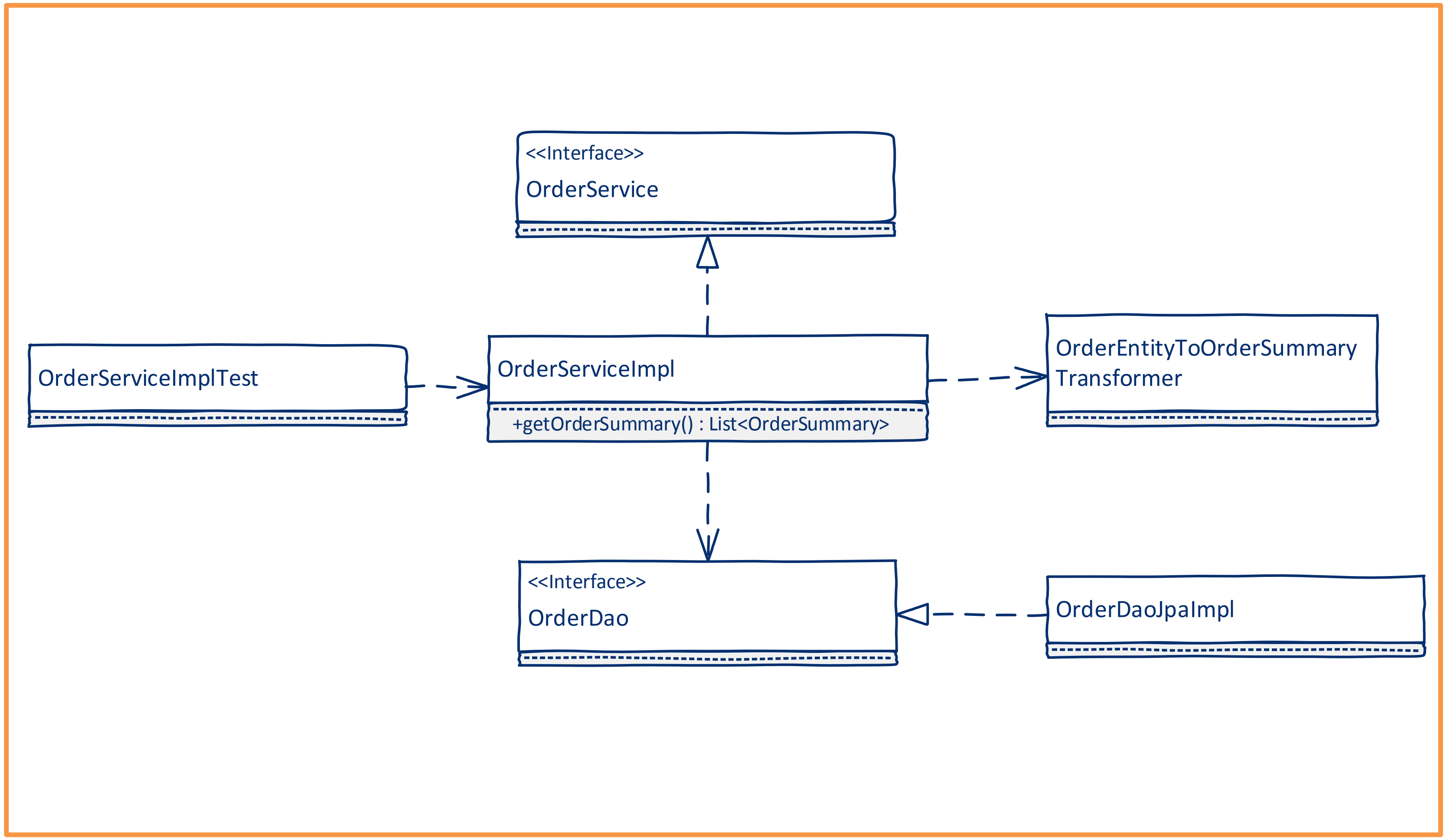
- General mocking concepts
- Introducing Mockito and a demonstration
- Core Mockito features & API

Mocking Concepts

Mocking Concepts

- Methods under test often leverage dependencies

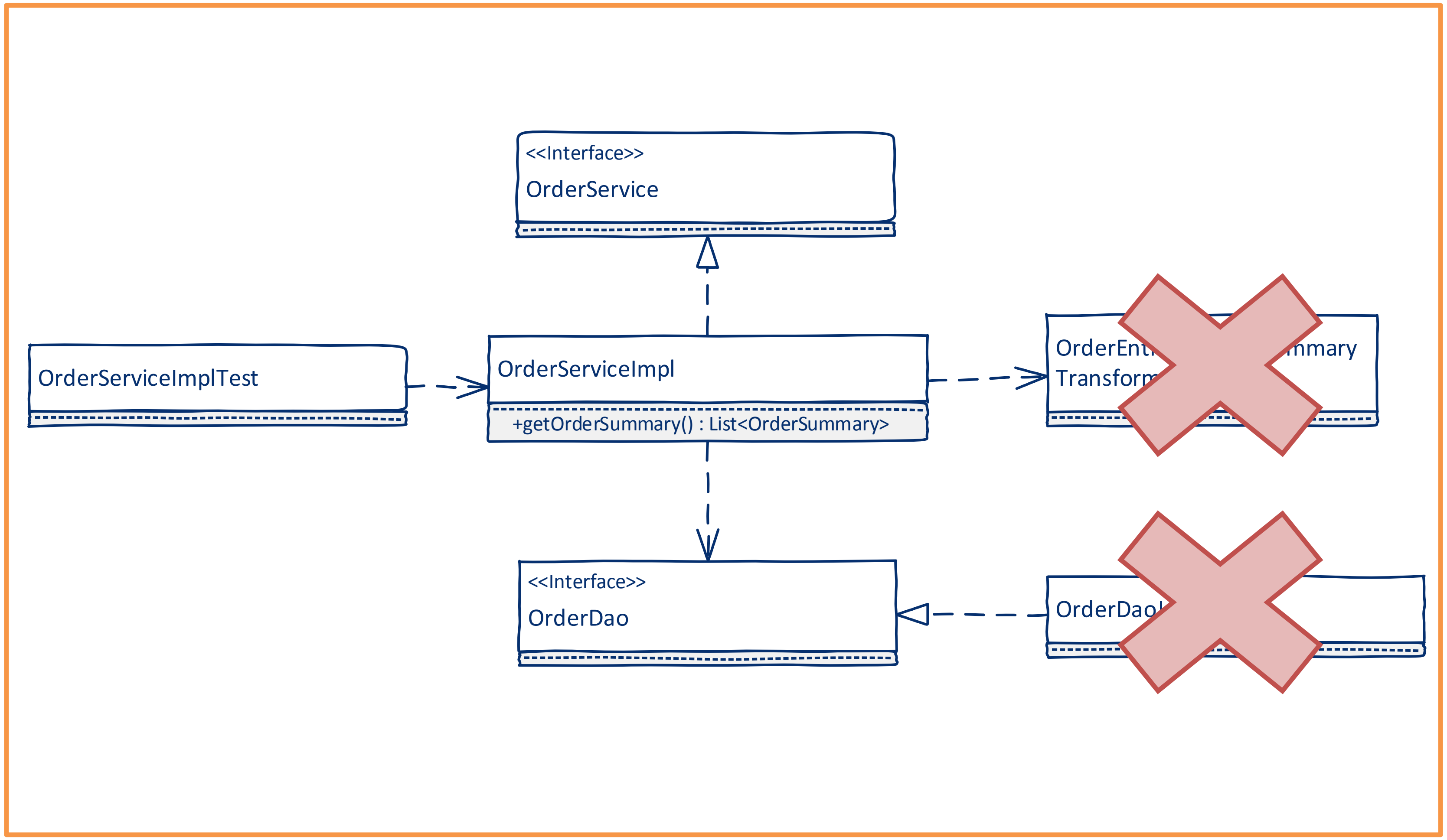
Mocking Concepts



Mocking Concepts

- **Methods under test often leverage dependencies**
- **Testing with dependencies creates challenges**
  - Live database needed
  - Multiple developers testing simultaneously
  - Incomplete dependency implementation
- **Mocking frameworks give you control**

Mocking Concepts



Mocking Options

- **Implement the mocked functionality in a class**
  - This approach is tedious and obscure

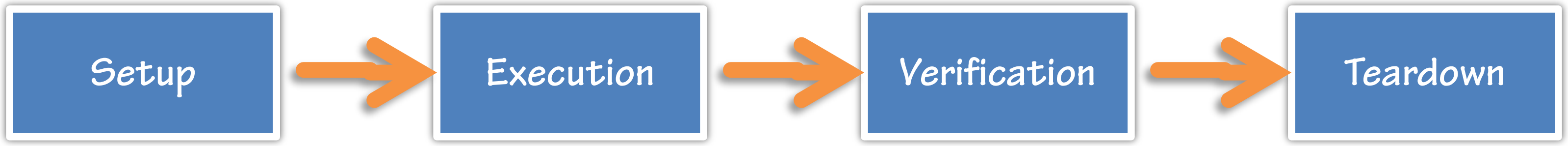
- **Leverage a mocking framework**
  - Avoid class creation
  - Leverages the proxy pattern

- **Multiple options – Mockito, EasyMock, JMock**

Mockito Overview

Mockito Overview

Support unit testing cycle



Setup – Creating the mock

OrderDao mockOrderDao = Mockito.*mock*(OrderDao.class)

Setup – Method stubbing

Mockito.*when*(mockOrderDao.findById(idValue)).*thenReturn*(orderFixture)

Verification

Mockito.*verify*(mockOrderDao).findById(idValue)

Mockito Demonstration

Creating Mock Instances

Creating Mock Instances

- Mockito.mock(*Class<?> class*) is the core method for creating mocks
- @Mock is an alternative

```
class OrderServiceTest {

  protected @Mock OrderDao mockOrderDao;

  @Before
  public void setup() {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void test_getOrderSummary {

    // You can use mockOrderDao
  }
}
```

MockSettings

- The MockSettings interface provides added control for mock creation
- Use MockSettings.extraInterfaces(...) to add interfaces supported by

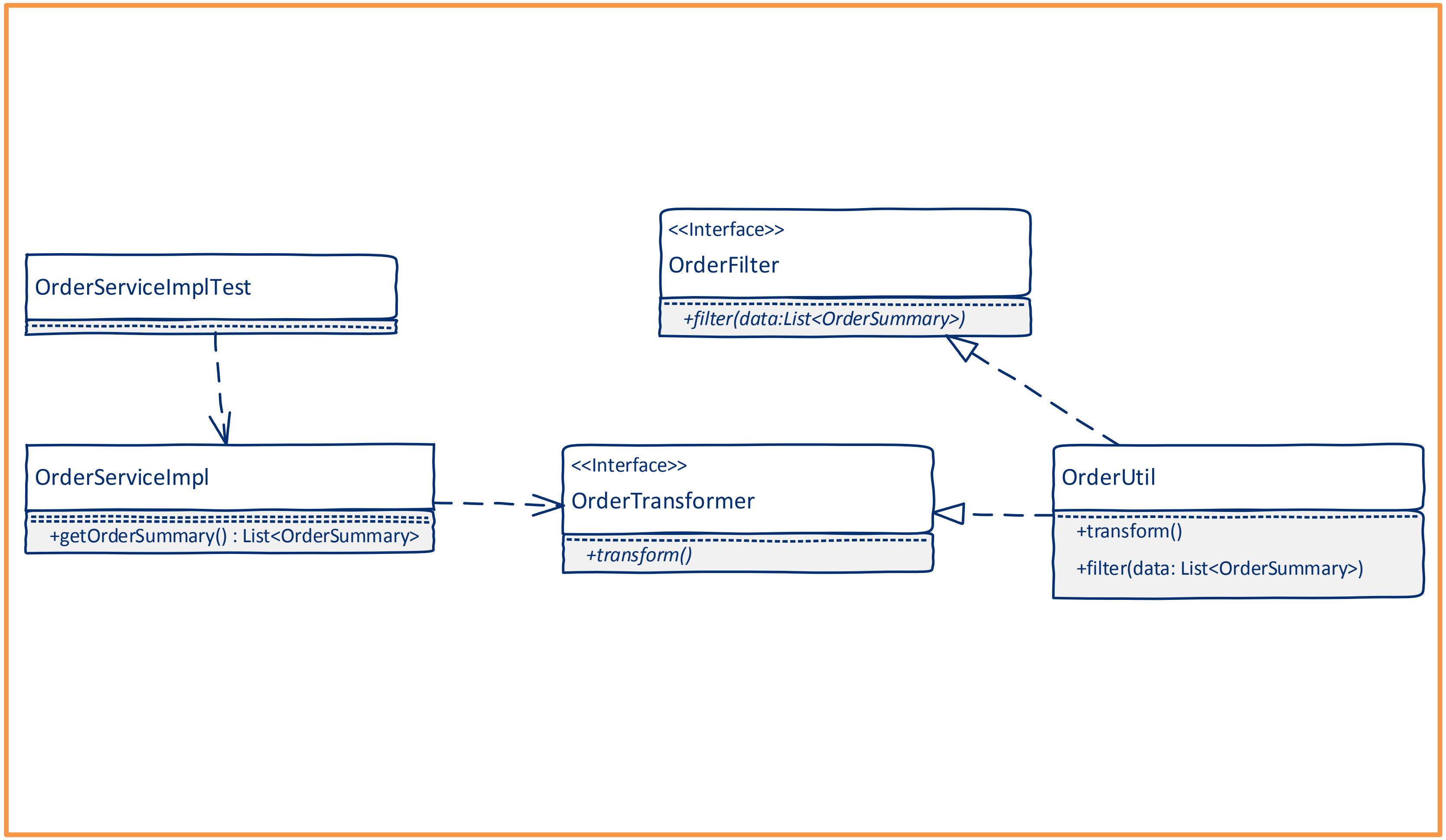
```
class OrderServiceImplTest {

  private OrderServiceImpl target = ...

  @Test
  public void test_getOrderSummary {

    MockSettings settings = Mockito.withSettings();
    OrderDao mockOrderDao = Mockito.mock(OrderDao.class, settings);
  }
}
```

MockSettings



MockSettings

```
class OrderServiceImplTest {

  private OrderServiceImpl target = . . .

  @Test
  public void test_getOrderSummary {

    MockSettings settings = Mockito.withSettings();
    OrderTransformer mockTransformer =
       Mockito.mock(OrderTransformer.class, settings.extraInterface(OrderFilter.class));

    target.getOrderSummary(mockTransformer);
  }
}
```

MockSettings

- The MockSettings interface provides added control for mock creation
- Use MockSettings.extraInterfaces(..) to add interfaces supported by the mock
- MockSettings.serializable() creates a mock which can be bassed as a serializable object
- MockSettings.name(..) specifies a name when verification of the mock fails

Stubbing Method Calls

Method Stubbing

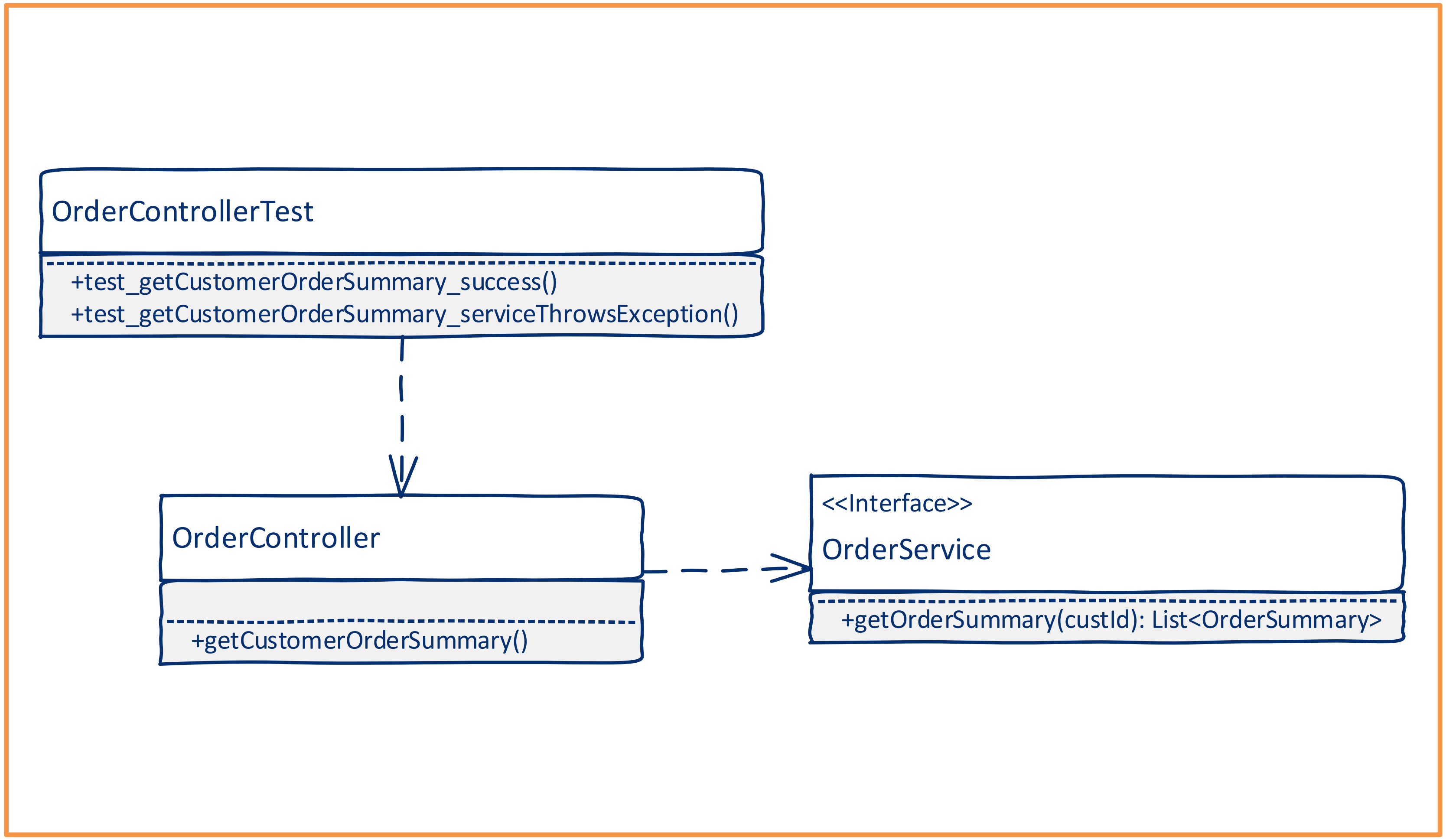
- **Provides capability to define how method calls behave via when/then pattern**

- **Calling Mockito.when(..) returns OngoingStub<T>, specifying how the invocation behaves**
  - thenReturn(..)
  - thenThrow(..)
  - thenCallRealMethod(..)
  - thenAnswer(..)

thenReturn(..)

- Specifies object or value to return when method called

thenReturn(..)


OrderControllerTest
+test_getCustomerOrderSummary_success()
+test_getCustomerOrderSummary_serviceThrowsException()
OrderController
+getCustomerOrderSummary()
<<Interface>>
OrderService
+getOrderSummary(custId): List<OrderSummary>

thenReturn(..)

- Specifies object or value to return when method called

```
class OrderControllerTest {

  private OrderController target;
  protected @Mock OrderService mockOrderService;
  // These members are initialized in a setup method

  @Test
  public void test_getCustomerOrderSummary_success {

    List<OrderSummary> orderSummaryFixtureList = . . .
    // Add fixture data to list
    . . .
    OngoingStub<List<OrderSummary>> invocationStub =
        Mockito.when(mockOrderService.getOrderSummary(customerId));
    invocationStub.thenReturn(orderSummaryListFixture);
    . . .
  }
}
```

thenReturn(..)

- Specifies object or value to return when method called

```
class OrderControllerTest {

  private OrderController target;
  protected @Mock OrderService mockOrderService;
  // These members are initialized in a setup method

  @Test
  public void test_getCustomerOrderSummary_success {

    List<OrderSummary> orderSummaryFixtureList = . . .
    // Add fixture data to list
    . . .
    Mockito.when(mockOrderService.getOrderSummary(customerId))
          .thenReturn(orderSummaryListFixture);
    . . .
  }
}
```

thenThrow(..)

- Specifies mock invocation should result in exception thrown

```
class OrderControllerTest {

   private OrderController target;
   protected @Mock OrderService mockOrderService;
   // These members are initialized in a setup method

   @Test
   public void test_getCustomerOrderSummary_serviceThrowsException {

      List<OrderSummary> orderSummaryFixtureList = ...
      // Add fixture data to list
      ...
      Mockito.when(mockOrderService.getOrderSummary(customerId))
            .thenThrow(new OrderServiceException("Test error reason"));
      ...
   }
}
```

void Methods

- Mocking void methods do not work with OngoingStub<T>
- Mockito.doThrow(..) returns the Stubber class

```
class OrderControllerTest {

  private OrderController target;
  protected @Mock OrderService mockOrderService;
  // These members are initialized in a setup method

  @Test
  public void test_processOrderSubmission_serviceThrowsException {

    Order orderFixture = …
    // Add fixture data to list
    …
    Stubber stubber = Mockito.doThrow(new OrderServiceException("Test error reason"))
    stubber.when(mockOrderService.processOrder(orderFixture));
    …
  }
}
```

Other Stubbing Response Options

- When mocking a class, delegate calls to the underlying instance with thenCallRealMethod(..)

```
Mockito.when(mockObject.targetMethod()).thenCallRealMethod();
```

- Answering allows you to provide a means to conditionally respond based on mock operation parameters

```
Mockito.when(mockObject.targetMethod(Mockito.any(String.class))).thenAnswer(new Answer() {

  Object answer(InvocationOnMock invocation) {
    ...
  }
});
```

Verifications

Verifications

- **Mockito.verify(..) is used to verify an intended mock operation was called**

```
//Setup
Mockito.when(mockOrderService.getOrderSummary(customerId)).thenReturn(orderList);

//Verification
Mockito.verify(mockOrderService).getOrderSummary(customerId);
```

Verifications

- Mockito.verify(..) is used to verify an intended mock operation was called

- VerificationMode allows extra verification of the operation

```
//Setup
Mockito.when(mockOrderService.getOrderSummary(customerId)).thenReturn(orderList);

//Verification
Mockito.verify(mockOrderService, VerificationSettings.times(2)).getOrderSummary(customerId);
```

Verifications

- **Mockito.verify(..) is used to verify an intended mock operation was called**

- **VerificationMode allows extra verification of the operation**
  - times(*n*)
  - atLeastOnce()
  - atLeast(*n*)
  - atMost(*n*)
  - never()

- **Verifying no interactions globally**
  - Mockito.verify(..).zeroInteractions()
  - Mockito.verify(..).noMoreInteractions()

Summary

- **Mocking concepts**
- **Mockito basic features**
  - Overview & Demo
  - Setup
  - Verification