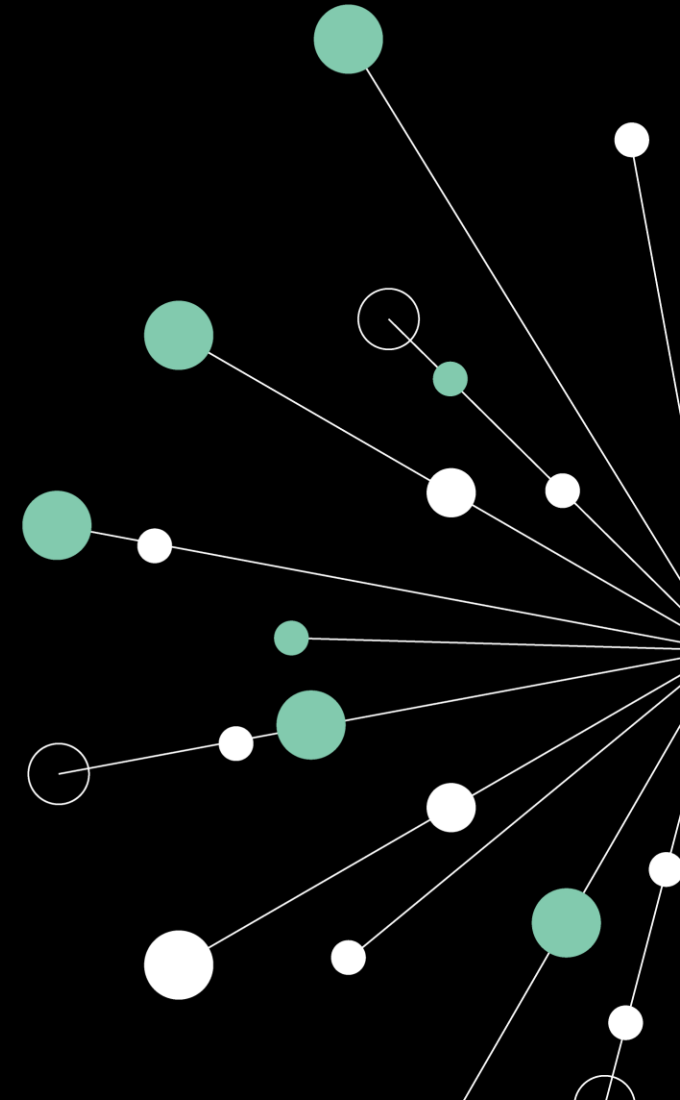


Testing code

Test Doubles





WHAT ARE TEST DOUBLES?



- **Your code interacts with other things**
 - Network resource – database, web service, etc.
 - Code being developed in parallel by other teams
- **Collaborators:**
objects with which the CUT interacts
- Use a **test double** if the collaborator doesn't exist yet or available

Test doubles allow you to...

- Run the test in its real environment
- Test interactions between your class and the other parts



WHAT IS MOCKING?

- A method for testing your code functionality in isolation
- **Does not require:**
 - Database connections
 - File server reads
 - Web services
- The mock object mocks the real service, returning dummy data
- Simulates the behaviour of a real external component





MOCK OBJECTS

Mock: ‘an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests’

– Massol, p. 141

To mock a database:

- You are not creating an object with state, you are creating an object which will respond to method calls as if it had a certain state

Several Dynamic Mock Frameworks in Java:

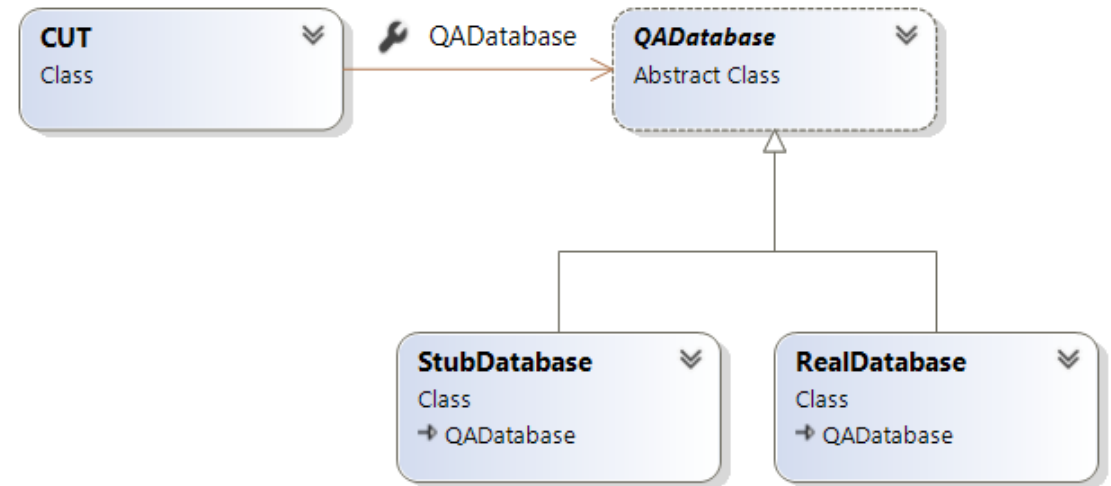
- EasyMock
- JMock
- Mockito
- Powermock
- JMockit and more



WHEN TO USE MOCK OBJECTS

When the real object...

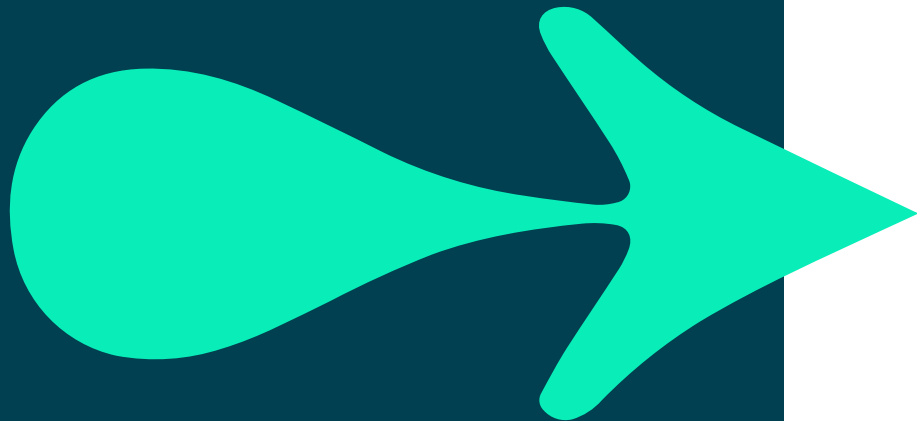
- Has non-deterministic behaviour
- Is difficult to set up
- Has behaviour that is hard to code (such as network error)
- Is slow
- Has (or is) a UI
- Uses a **callback** : tests need to query the object, but the queries are not available in the real object (for example, 'was this callback called?')
- Does not yet exist





DRAWBACKS OF MOCK OBJECTS

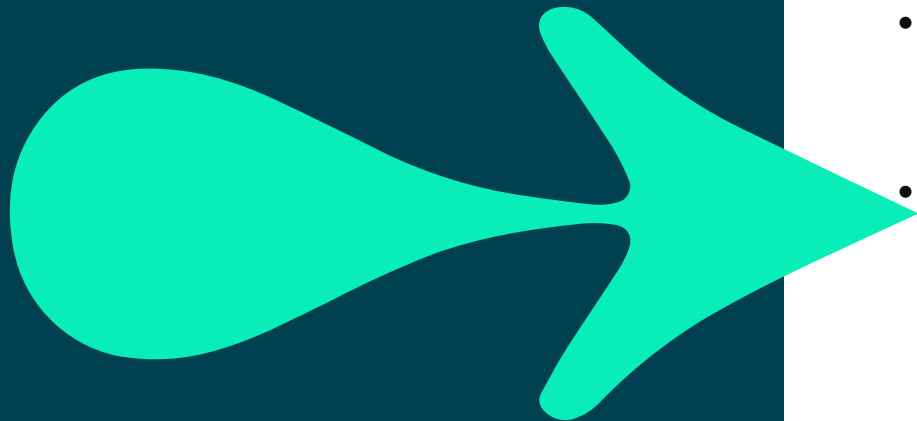
1. Mocks don't test interactions with container or between the components.
2. Tests are coupled very tightly to implementation.
3. Mocks don't test the deployment part of components.
4. Most frameworks can't mock static methods, final classes and methods





MOCKS ADVANTAGES

- **Test components independently and Don't depend on external factors**
problems with network latency or third-party downtimes
- **Tests run faster** because no actual resource is used
- **Precise control** over the behaviour by Simulating specific scenarios like errors or edge cases
- **Cost effective:** don't have to se up systems
- **When external dependencies are not yet available**





MOCKS DISADVANTAGES

- **False Confidence:**
Could ignore issues with integration or production.
- **Maintenance:**
You must maintain code when the actual system changes
- **Complexity:** Maintenance may become complex
- **Realism:**
May ignore performance, bottlenecks or unexpected API changes and may miss issues that arise only during real-world integration.
- **Stale Tests:** If the real system evolves but the mocks don't, tests may pass while the code breaks in production



STUBS



Stub: controllable replacement for existing dependency

- A class which is (ideally) the simplest possible implementation of the logic in the real code
- Provide pre-programmed answers to calls they receive
- Won't respond to anything outside of what you've programmed

Good for:

- Coarse-grained testing, e.g. replacing a complete external system

Drawbacks:

- Often complex to write
- Introduce their own debugging & maintenance issues



MOCKS VS STUBS

- **Not mutually exclusive**; you can combine their use
- Both can be handwritten, simple classes – simplicity and readability are at an advantage over frameworks
- **Stubs** enable tests by replacing external dependencies
- **Asserts** are against the Class Under Test, not the stub

Test <————> **CUT** <————> Stub
 asserts interacts



MOCKS VS STUBS

- Using a **mock** is much like using a stub, except that the mock will **record the interactions**, which can be verified
- **Asserts** are verified against the mock

CUT <————> **Mock** <————> Test
interacts asserts

Injecting dependencies removes tight coupling – an example

```
//Interface for abstraction
public interface IEngine {
    void start();
}
```

```
public class SlowEngine implements IEngine {
    @Override
    public void start() {
        System.out.println("Slow engine started.");
    }
}
```

```
public class FastEngine implements IEngine {
    @Override
    public void start() {
        System.out.println("Fast engine started.");
    }
}
```

```
public class Car {
    IEngine engine;

    // Dependency injected via constructor
    public Car(IEngine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
    }
}
```

```
public static void main(String[] args) {
    IEngine engine = new FastEngine(); // or SlowEngine();
    Car car = new Car(engine); // Inject
    car.drive();
}
```



DEPENDENCY INJECTION CONTAINERS

- Containers inject the mock objects into unit tests during unit testing
- Some DI containers include:
 - Spring DI
 - Butterfly DI Container
 - Dagger
 - Guice
 - PicoContainer
- Many unit tests don't require a DI container if their dependencies are simple to mock out

Lab

"Test Doubles" Lab

Code examples are offered in your next lab but for more examples of using **mocks** and **stubs** please see slides after this page





Revision Creating stubs with C#



Stub – Using C# interface/Abstract class



```
public interface IQA_Database {  
    Product getProduct(int id);  
}
```

```
public class StubQADatabase : IQA_Database {  
    Product[] products = { new Product(10, "Pen", 50),  
                           new Product(80, "Ruler", 1.20), new Product(99, "Paper", 30.0) };  
  
    public Product getProduct(int id) {  
        return Product product = Arrays.stream(products)  
            .filter(p -> p.getId() == 10).findFirst().orElse(null);  
    }  
}
```



```
public class Order {  
    public bool ProcessOrders(IQA_Database database, int productID, int quantity) {  
        Product product = database.getProduct(productID);  
        /*  
        *   Code to process the order. Code under test  
        */  
    }  
    return true;  
}
```

```
@Test  
public void TestOrderProcessReturnsTrueForValidProductID() {  
    Order order = new Order();  
    Assert.IsTrue(order.ProcessOrders(new StubQADatabase(), 80, 5);  
}
```

Using TestInitialize - Stubs with MS-Test



```
[TestClass]
public class QATests {

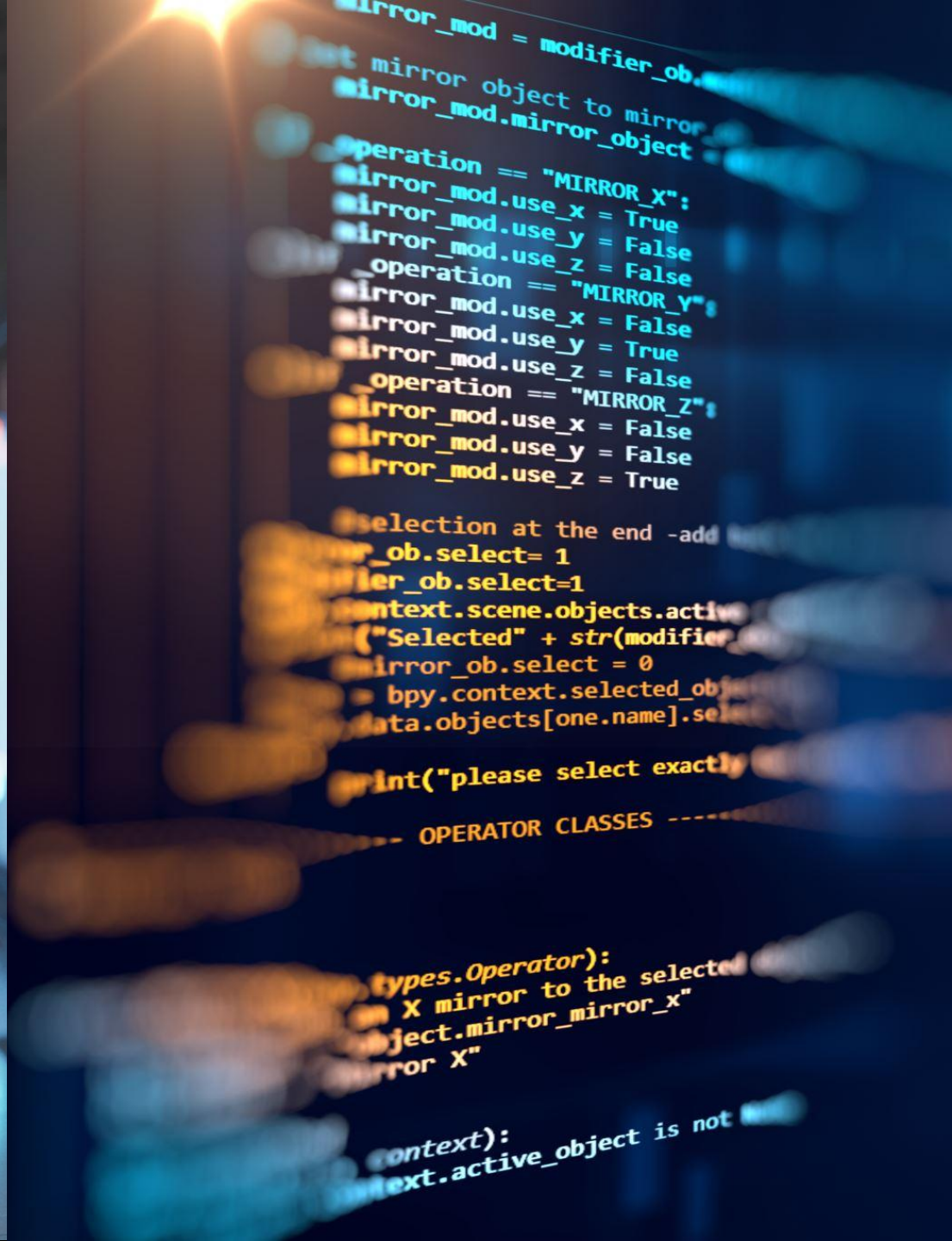
    IQA_Database database;

    [TestInitialize]
    public void Setup()
    {
        database = new StubQADatabase();
    }

    [TestMethod]
    public void TestOrderProcessReturnsTrueForValidProductID()
    {
        Order order = new Order();
        Assert.IsTrue(order.ProcessOrders(database, 80, 5);
    }
}
```



Creating stubs with Java



Stub – Using Java interface/Abstract class



```
public interface IQA_Database {  
    Product getProduct(int id);  
}
```

```
public class StubQADatabase implements IQA_Database {  
    Product[] products = { new Product(10, "Pen", 50),  
                           new Product(80, "Ruler", 1.20), new Product(99, "Paper", 30.0)};  
  
    public Product getProduct(int id) {  
        return // code to find and return a product with the given id  
    }  
}
```



```
public class Order {  
    public boolean ProcessOrders(IQA_Database database, int productID, int quantity) {  
        Product product = database.getProduct(productID);  
        /*  
        *   Code to process the order. Code under test  
        */  
    }  
    return true;  
}
```

```
@Test  
public void TestOrderProcessReturnsTrueForValidProductID() {  
    Order order = new Order();  
    AssertTrue(order.ProcessOrders(new StubQADatabase(), 80, 5);  
}
```

Using @BeforeEach - Stubs with JUnit



```
public class QATests {  
  
    IQA_Database database;  
  
    @BeforeEach  
    public void setup()  
    {  
        database = new StuQADatabase();  
    }  
  
    @Test  
    public void TestOrderProcessReturnsTrueForValidProductID()  
    {  
        Order order = new Order();  
        AssertTrue(order.ProcessOrders(database, 80, 5);  
    }  
}
```

Testing code

Mocks





Creating mocks with C#



Revision - Creating a Stub using C#



```
public interface IQA_Database {  
    Product getProduct(int id);  
}
```

```
public class StubQADatabase : IQA_Database {  
    Product[] products = { new Product(10, "Pen", 50),  
                           new Product(80, "Ruler", 1.20), new Product(99, "Paper", 30.0) };  
  
    public Product getProduct(int id) {  
        return // code to find and return a product with the given id  
    }  
}
```



```
public class Order {  
    public bool ProcessOrders(IQA_Database database, int productID, int quantity) {  
        Product product = database.getProduct(productID);  
        /*  
        *   Code to process the order. Code under test  
        */  
    }  
    return true;  
}
```

```
@Test  
public void TestOrderProcessReturnsTrueForValidProductID() {  
    Order order = new Order();  
    Assert.IsTrue(order.ProcessOrders(new StubQADatabase(), 80, 5);  
}
```

C#: Using Moq



```
[TestMethod]
public void TestOrderProcessReturnsTrueForValidProductID()
{
    var moq = new Mock<IQA_Database>();

    moq.Setup(x => x.getProduct(10)).Returns(new Product(10, "Pen", 50), );
    moq.Setup(x => x.getProduct(80)).Returns(new Product(80, "Ruler", 1.20));
    moq.Setup(x => x.getProduct(99)).Returns(new Product(99, "Paper", 30.0) );

    var order = new Order();
    bool res = order.ProcessOrders(moq.Object, 80, 5);
    Assert.IsTrue(res);
}
```

```
public interface IQA_Database {
    Product getProduct(int id);
}
```

Using TestInitialize



```
[TestClass]
public class Temp {

    IQA_Database database;

    [TestInitialize]
    public void Setup()
    {
        var moq = new Mock<IQA_Database>();

        moq.Setup(x => x.getProduct(10)).Returns(new Product(10,"Pen",50), );
        moq.Setup(x => x. getProduct(80)).Returns(new Product(80,"Ruler",1.20));
        moq.Setup(x => x. getProduct(99)).Returns(new Product(99,"Paper",30.0) );
        database = moq.Object
    }

    [TestMethod]
    public void TestOrderProcessReturnsTrueForValidProductID()
    {
        Order order = new Order();
        bool res = order.ProcessOrders(database, 80, 5);
        Assert.IsTrue(res);
    }
}
```

```
public interface IQA_Database {
    Product getProduct(int id);
}
```

MOQ – Type of Any and Exception



```
public interface Ilogin {  
    bool login(string userName, string password, int userType);  
}
```

```
ILogin login;
```

```
[TestInitialize]
```

```
public void Setup() {  
    Mock<ILogin> mockLogin = new Mock<ILogin>();  
    mockLogin.Setup(x => x.login("Bob", "Password123", It.IsAny<int>())).Returns(true);  
    mockLogin.Setup(x => x.login("Steve", "sa", 2)).Returns(true);  
  
    mockLogin.Setup(x => x.login(It.IsAny<string>(), It.IsAny<string>(),  
                                It.IsNotIn(1, 2))).  
                Throws<ArgumentException>();  
  
    login = mockLogin.Object;  
}
```

```
[TestMethod]
```

```
public void TestRegisterEmployeeWithValidUserType() {  
    var qa = new Company();  
    Assert.IsTrue( qa.registerEmployee("Bob", "password123", login, 1) );  
}
```

MOQ – Verify a method was called



```
public interface IEmailService {  
    void SendEmail(string recipient, string subject, string body);  
}
```

```
public class OrderProcessor  
{  
    private IEmailService emailService;  
  
    public OrderProcessor(IEmailService emailService) {  
        this.emailService = emailService;  
    }  
  
    public void ProcessOrder(string customerEmail, double price) {  
        // Order processing logic ...  
        if(price > 100)  
            emailService.SendEmail(customerEmail, "Confirmation", "Order processed");  
    }  
}
```

MOQ – Verify...



```
[TestMethod]
public void ProcessOrder_HighPrice_Should_SendEmail()
{
    var mockEmailService = new Mock<IEmailService>();
    var orderProcessor = new OrderProcessor(mockEmailService.Object);

    orderProcessor.ProcessOrder("Bob@QA.com", 200);

    mockEmailService.Verify(
        e => e.SendEmail("Bob@QA.com", "Confirmation", "Order processed"), Times.Once );
}
```

MOQ – Verify...



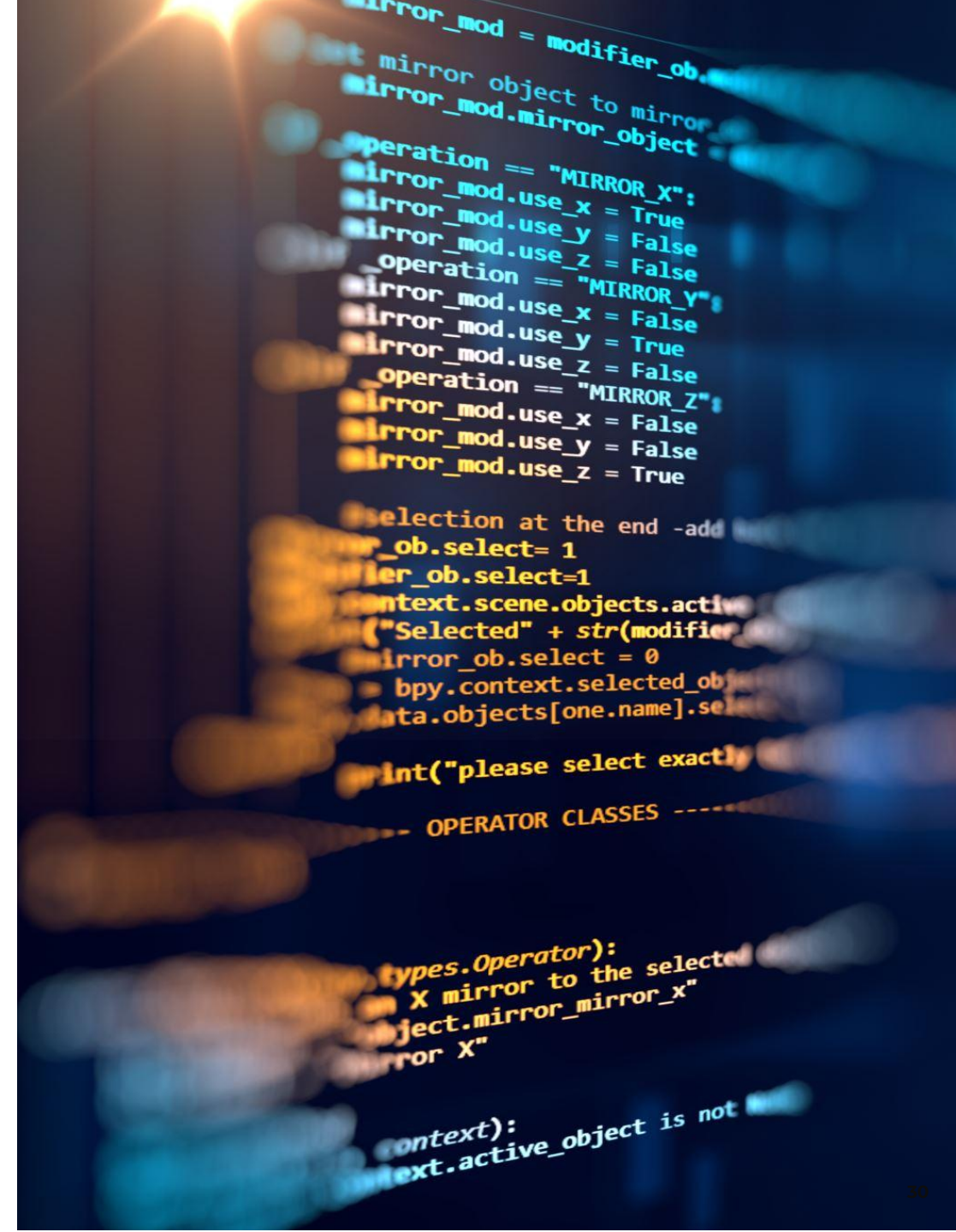
```
[TestMethod]
public void ProcessOrder_HighPrice_Should_SendEmail()
{
    var mockEmailService = new Mock<IEmailService>();
    var orderProcessor = new OrderProcessor(mockEmailService.Object);

    orderProcessor.ProcessOrder("Bob@QA.com", 10);

    mockEmailService.Verify(
        e => e.SendEmail("Bob@QA.com", "Confirmation", "Order processed"), Times.Never );
}
```



Creating mocks with Java



Using Mochito with Java



- Mockito is a useful app to mock external dependencies.
- To use **Mochito**, you need to set it up. Here is a POM file example

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.1.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Mockito - Setting expectations



- Creates an interface for a dependency in your Unit tests
- Set up your method's expected values
- So, how do we get Mockito to do all these? Let's see...

Mocking an object using Mockito



```
public class QADatabase {  
    public Product getProduct(int id){  
        // complex operation to get the product from a real database  
    }  
}
```

Object
to mock

```
public class QATest {  
    QADatabase db;  
  
    @BeforeEach  
    public void setUp() {  
        db = Mockito.mock(QADatabase.class);  
        Mockito.when(db.getProduct(10)).thenReturn(new Product(10, "Pen", 50));  
        Mockito.when(db.getProduct(80)).thenReturn(new Product(80, "Ruler", 1.20));  
        Mockito.when(db.getProduct(99)).thenReturn(new Product(99, "Paper", 30.0));  
    }  
}
```

Mokito Exceptions



```
public class QATest {
    Account acc;

    @BeforeEach
    public void setup() throws IllegalArgumentException
    {
        acc = Mockito.mock(Account.class);
        Mockito.when(acc.withdraw(200.0)).thenThrow(IllegalArgumentException.class);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testAccount_WithdrawingInsufficientFundsThrows_IllegalArgumentException()
    {
        // many lines of code leading to x = 200;

        acc.withdraw(x);
    }
}
```

Mochito Verify



```
public interface EmailService {  
    void sendEmail(String recipient, String subject, String body);  
}  
  
public class OrderProcessor {  
    private EmailService emailService;  
  
    public OrderProcessor(EmailService emailService) {  
        this.emailService = emailService;  
    }  
  
    public void processOrder(String customerEmail, double price) {  
        // Order processing logic...  
        // Send confirmation email  
        if(price > 100)  
            emailService.sendEmail(customerEmail, "Confirmation", "Order processed.");  
    }  
}
```

Mochito - Verify



```
@Test
void processOrder_ShouldCall_SendEmail()
{

    // Arrange
    EmailService mockEmailService = Mockito.mock(EmailService.class);
    OrderProcessor orderProcessor = new OrderProcessor(mockEmailService);

    // Act
    orderProcessor.processOrder("Bob@QA.com", 200);

    // Assert
    verify(mockEmailService, times(1))    // times(0) or never() if not called
        .sendEmail("Bob@QA.com", "Confirmation", "Order processed.");

}
```

- Exercise

Summary

- In this section you revised how to create and use stubs & mocks

