



University of
Sheffield



COM3529 Software Testing and Analysis

Unit Testing – Part 2

Professor Phil McMinn

How to Write Clear Unit Tests

JUnit v. Hamcrest Assertions

The default supplied JUnit assertions have some deficiencies:

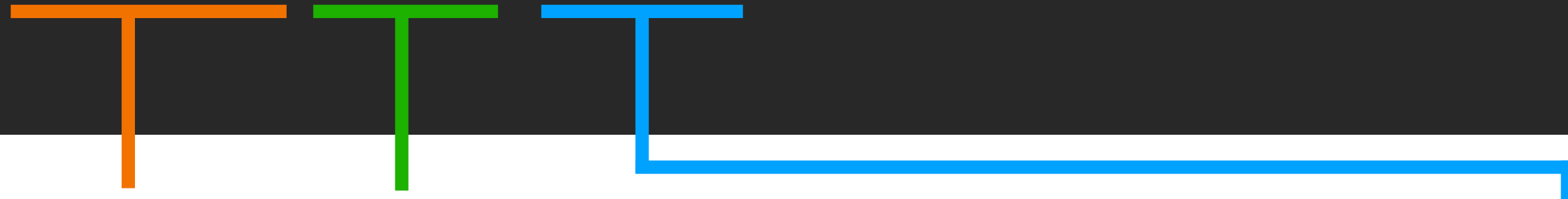
- **It's easy with the assertion method parameters to get expected and actual the wrong way round.** (I do it all the time!) It's not terribly consequential, which is why it's easy to do, but the wrong ordering will confuse other programmers.
- **A different style is required for differing expected-actual relationships – i.e. a different assertion method**
- The different assertion methods available are somewhat limited
- It's difficult to customise error messages

For these reasons, some programmers prefer the Hamcrest style of assertion.

Did You Notice We'd Switched to a Different Style of Assertion?

Hamcrest Assertions

```
@Test
public void isocetesTest() {
    Triangle.Type result = Triangle.classify(5, 10, 10);
    assertThat(result, equalTo(Triangle.Type.ISOSCELES));
}
```



Every assertion uses the generic **assertThat** method

The general assertion format maps more closely to natural language, making it more obvious that it's the **actual result** of the unit under test that goes first in the parameter order.

The relationship between actual and expected results is specified by a **matcher**, in this case, **equalTo**.

Hamcrest Matchers

Hamcrest has a plethora of “matchers”, like `equalTo`.

They can help write assertions involving a variety of types, including:

- Strings – e.g., can check if a string contains a substring, ignore case etc.
- Collections – whether an element is in a collection; what a collection contains, ignoring order etc.
- See <http://hamcrest.org/JavaHamcrest/javadoc/2.2/org/hamcrest/Matchers.html>

If the appropriate matcher is not available it's very easy to write your own.

See <https://www.baeldung.com/java-junit-hamcrest-guide>

Make Your Tests *Complete* and *Concise*

Ensure the test contains all the information needed for a reader to understand how it arrived at its result.

Ensure it contains no other irrelevant and distracting information.

A test case is *complete* when its body contains all of the information a reader needs to understand how it arrives at its result.

```
@Test
// An incomplete test!
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    makeMoves(c4);
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

What's going on in this helper method?

Why RED? Where does that come from?

Make Your Tests *Complete*

A test case is **complete** when its body contains all of the information a reader needs to understand how it arrives at its result.

```
@Test
// An incomplete test!
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    makeMoves(c4);
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

What's going on in this helper method?

Why **RED**? Where does that come from?

Make Your Tests *Complete*

A test case is **complete** when its body contains all of the information a reader needs to understand how it arrives at its result.

```
@Test
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Enumerating all the moves makes the method longer and prevents re-using the move sequence as a helper method, BUT makes it clearer what's going. Two columns of vertical pieces are being added to the board, and **RED** wins in column 0

Don't DRY Tests

DRY – Don't Repeat Yourself: engineer needs to update one piece of code rather than tracking down all instances.

Downside: Can make code less clear, **requires following chains of references.** This might be a small price to pay for making the code easier to work with...

... but the cost/benefit analysis plays out differently with tests:

- **We want tests to break when software changes**
- **Production code has the benefit of a test suite to ensure it keeps working when things get complex. Tests should stand on their own!**
(Something has gone wrong when tests need tests)

DAMP not DRY

DAMP: Descriptive And Meaningful Phrases

DAMP is not a *replacement* for DRY – it is complementary.

“Helper” methods can help make tests clearer by making them more concise – factoring out repetitive steps whose details aren’t relevant to the behaviour being tested.

But the refactoring should be done with an eye for make the tests more readable and descriptive, not solely to reduce repetition.

Don’t comprise clarity and conciseness.

Make Your Tests *Concise*

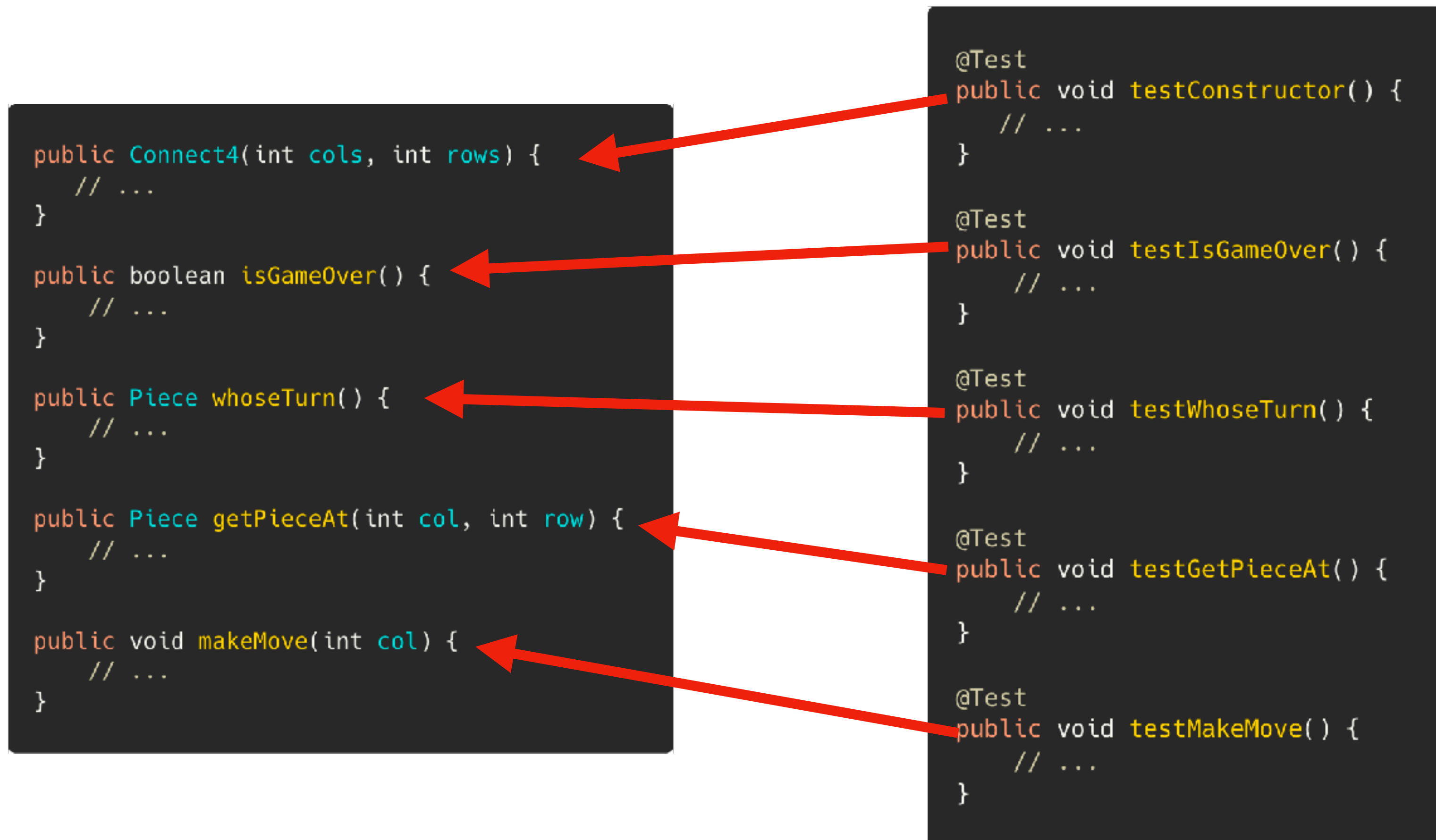
A test case is **concise** when it contains no other distracting or irrelevant information.

```
@Test
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(2); // RED
    c4.makeMove(2); // YELLOW
    c4.makeMove(3); // RED
    c4.makeMove(3); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Our test now includes a lot of moves that are not needed for the test scenario and make the board even harder to visualise and what the test outcome should be.

Don't Test Methods – Test Behaviours

The first instinct of many engineers is to try to match the structure of their tests to the structure of the code.



Don't Test Methods – Test Behaviours

But a single method may have more than one behaviour and/or some tricky corner cases that require more tests.

For example, `makeMove` can have several behaviours, depending on the state of the board.

One test for that method makes no sense, and is likely to not be very clear nor concise.

Better way: Write tests for each behaviour.

Testing Behaviour

A behaviour is a guarantee that a system makes about how it will respond to a series of inputs while in a particular state.

A behaviour can be expressed with “**given X when Y, then Z**”

For example:

Given a Connect4 board, with RED starting first

When RED has played a piece

Then it's YELLOW's turn next.

Writing Behaviour-Driven Tests

Behaviour-Driven tests tend to read more like natural language, so structure them accordingly.

```
@Test
public void shouldChangePieceAfterTurn() {
    // Given a Connect 4 Board, with RED starting first
    Connect4 c4 = new Connect4(7, 6);

    // When RED makes a move
    c4.makeMove(0);

    // Then it's YELLOW's turn next
    assertThat(c4.whoseTurn(), equalTo(Piece.YELLOW));
}
```

Name Tests after the Behaviour Being Tested

A good name describes the actions (the “when”) that are being tested and the expected outcome (the “then”), and sometimes the state to (the “given”).

A good trick is to start the name with “should”, e.g.

`shouldInitializeCorrectly`
`shouldChangePieceAfterTurn`
`shouldEndGameWithWinnerWhenFourInARowHorizontally` etc.

Don't Put Logic in Tests

Don't put conditionals or loops in tests, or logical operations.

Tests should read as simple statements of truth, *not chunks of code that also require tests!*

Here the test is trying to check every position of the board and ensure it is not set to a piece (i.e., it is null)

```
@Test
public void shouldInitializeCorrectly() {
    // Given a new Connect 4 Board
    Connect4 c4 = new Connect4(7, 6);

    // Then it's RED's turn
    assertEquals(c4.whoseTurn(), equalTo(Piece.RED));

    // Then the board has no piece in every position
    for (int i=0; i < 7; i++) {
        for (int j=0; j < 6; j++) {
            assertEquals(c4.getPieceAt(j, j), nullValue());
        }
    }

    // Then the game is not over
    assertEquals(c4.isGameOver(), equalTo(false));

    // Then there is no winner
    assertEquals(c4.winner, equalTo(null));
}
```

But oops, the developer made a mistake, checking `(j, j)` instead of `(i, j)`. The test won't fail, so the mistake is hard to spot.

Don't Put Logic in Tests

Don't put conditionals or loops in tests, or logical operations.

Tests should read as simple statements of truth, *not chunks of code that also require tests!*

Better just to initialise a smaller 2x2 board and explicitly check each position

```
@Test
public void shouldInitializeCorrectly() {
    // Given a new Connect 4 Board
    Connect4 c4 = new Connect4(2, 2);

    // Then it's RED's turn
    assertThat(c4.whoseTurn(), equalTo(Piece.RED));

    // Then the board has no piece in every position
    assertThat(c4.getPieceAt(0, 0), nullValue());
    assertThat(c4.getPieceAt(0, 1), nullValue());
    assertThat(c4.getPieceAt(1, 0), nullValue());
    assertThat(c4.getPieceAt(1, 1), nullValue());

    // Then the game is not over
    assertThat(c4.isGameOver(), equalTo(false));

    // Then there is no winner
    assertThat(c4.winner, equalTo(null));
}
```


Making Your Unit Tests Clear

- 1 Make your tests **concise** and **complete** (DAMP and not too DRY!)
- 2 Don't structure tests around methods – instead **structure around behaviours**
- 3 Use the **Given-When-Then** pattern for testing behaviour
- 4 **Name Tests after the Behaviour Being Tested**
- 5 **Don't put logic in tests**



University of
Sheffield



COM3529 Software Testing and Analysis

Test Doubles

Professor Phil McMinn

A Testing Problem

To: p.mcminn@sheffield.ac.uk
From: student3529@sheffield.ac.uk
Subject: A Problem with Testing – Please help!!!

Dear Phil

For my dissertation code, I want to test the logic in my class using a unit test, but the class also involves a database connection.

To include the database is kind of complex, and I do not want to test integration aspects at the same time as unit testing the logic.

What should I do?

Yours,
Stu

A Testing Solution

To: student3529@sheffield.ac.uk

From: p.mcminn@sheffield.ac.uk

Subject: Re: A Problem with Testing – Please help!!!

Dear Stu,

Have no fear.

You might want to consider using a **test double**.

We're going to cover this in the next lecture – be sure to be there!

Best,
Phil

Scenario

We have a class called **BankAccount**.

A database is used to store and retrieve bank account information.

How do we unit test the logic of this class without interacting with the actual database?

```
public class BankAccount {  
  
    private final int bankAccountNumber;  
    private final BankAccountDatabaseConnection bankAccountDatabaseConnection;  
  
    public BankAccount(BankAccountDatabaseConnection bankAccountDatabaseConnection,  
                       int openingBalance, int overdraft) {  
        this.bankAccountDatabaseConnection = bankAccountDatabaseConnection;  
        this.bankAccountNumber = bankAccountDatabaseConnection.createBankAccount();  
        setOverdraft(overdraft);  
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, openingBalance);  
    }  
  
    public void withdraw(int amount) {  
        if (amount <= 0) {  
            throw new BankAccountException("Cannot withdraw a zero or negative amount");  
        }  
        int balance = getBalance();  
        int maxWithdrawalAmount = balance + getOverdraft();  
        if (amount > maxWithdrawalAmount) {  
            throw new BankAccountException(  
                "Cannot withdraw more than the current balance plus the overdraft");  
        }  
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);  
    }  
  
    public void deposit(int amount) {  
        if (amount <= 0) {  
            throw new BankAccountException("Cannot deposit a zero or negative amount");  
        }  
        bankAccountDatabaseConnection.setBalance(bankAccountNumber, getBalance() + amount);  
    }  
  
    public void setOverdraft(int amount) {  
        if (amount < 0) {  

```


Scenario

We have a class called **BankAccount**.

A database is used to store and retrieve bank account information.

How do we unit test the logic of this class without interacting with the actual database?

```
}
    int balance = getBalance();
    int maxWithdrawalAmount = balance + getOverdraft();
    if (amount > maxWithdrawalAmount) {
        throw new BankAccountException(
            "Cannot withdraw more than the current balance plus the overdraft");
    }
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);
}

public void deposit(int amount) {
    if (amount <= 0) {
        throw new BankAccountException("Cannot deposit a zero or negative amount");
    }
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, getBalance() + amount);
}

public void setOverdraft(int amount) {
    if (amount < 0) {
        throw new BankAccountException("Overdraft cannot be negative");
    }
    bankAccountDatabaseConnection.setOverdraft(bankAccountNumber, amount);
}

public int getBankAccountNumber() {
    return bankAccountNumber;
}

public int getBalance() {
    return bankAccountDatabaseConnection.getBalance(bankAccountNumber);
}

public int getOverdraft() {
    return bankAccountDatabaseConnection.getOverdraft(bankAccountNumber);
}
}
```

The Problematic Object

We want to test the logic in **BankAccount** without having to worry about the underlying database that it uses ... implemented by this class – which sends SQL queries to the database and returns values contained in it.

```
public class BankAccountDatabaseConnection {  
  
    public int createBankAccount() {  
        // make database calls  
    }  
  
    public int getBalance(int bankAccountNo) {  
        // make database calls  
    }  
  
    public void setBalance(int bankAccountNo, int amount) {  
        // make database calls  
    }  
  
    public int getOverdraft(int bankAccountNo) {  
        // make database calls  
    }  
  
    public void setOverdraft(int bankAccountNo, int amount) {  
        // make database calls  
    }  
  
}
```


Test Doubles

Test Doubles are classes that “stand in” for some original class, allowing tests to avoid some of the complexity needed if that original class had been used instead.

Test doubles are a bit like **stunt doubles** – instead of using the real actor, we use another that looks like it but makes all the tough stuff look easy!



Ryan Gosling and his stunt double on the set of the “Barbie” movie.

Types of Test Double

1 Dummies

2 Stubs

3 Fakes

4 Mocks

5 Spies

Dummies

Dummies are objects that stand in place of the real object.

However the test never makes use of the dummy, its purpose is just to satisfy the compiler.

In this and the following slides, we assume **BankAccountDatabaseConnection** is a Java interface that we can implement in different ways for testing. But the methods of a real class could easily be overridden to achieve the same effect.

```
public class DummyBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 0;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
    }

    @Override
    public int getOverdraft(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setOverdraft(int bankAccountNo, int amount) {
    }
}
```

Dummy

Dummies

Method
under test

The database
itself does not
matter for the
purposes of this
test. So we just
need a dummy
to get the test
to compile.

Test
using
dummy

```
public void withdraw(int amount) {  
    if (amount <= 0) {  
        throw new BankAccountException("Cannot withdraw a zero or negative amount");  
    }  
    int balance = getBalance();  
    int maxWithdrawalAmount = balance + getOverdraft();  
    if (amount > maxWithdrawalAmount) {  
        throw new BankAccountException(  
            "Cannot withdraw more than the current balance plus the overdraft");  
    }  
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);  
}
```

```
@Test  
public void shouldNotAllowNegativeAmountsToBeWithdrawn() {  
    DummyBankAccountDatabaseConnection dummy = new DummyBankAccountDatabaseConnection();  
  
    // Given a bank account  
    BankAccount bankAccount = new BankAccount(dummy, 0, 0);  
  
    // When a negative amount is withdrawn, Then an exception is thrown  
    assertThrows(BankAccountException.class, () -> {  
        bankAccount.withdraw(-1000);  
    });  
}
```

Stubs

Stubs are objects that override certain methods of the original so that some other class/method can be tested.

```
public BankAccount(BankAccountDatabaseConnection bankAccountDatabaseConnection,
                    int openingBalance, int overdraft) {
    this.bankAccountDatabaseConnection = bankAccountDatabaseConnection;
    this.bankAccountNumber = bankAccountDatabaseConnection.createBankAccount();
    setOverdraft(overdraft);
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, openingBalance);
}
```

BankAccount –
original constructor

```
public class StubbedBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 1000;
    }

    // ...
}
```

Stubbed method of
BankAccountDatabaseConnection

Test
using
stub

```
@Test
public void shouldAssignABankAccountWhenOpeningAnAccount() {
    StubbedBankAccountDatabaseConnection stub = new StubbedBankAccountDatabaseConnection();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(stub, 0, 0);

    // Then it should have a bank account number
    assertThat(bankAccount.getBankAccountNumber(), equalTo(1000));
}
```

Fakes

Fakes provide pseudo-implementations of the real object.

Here – an “in-memory” implementation of the database functionality.

Note the downside of fakes – essentially we’re implementing more functionality that itself needs testing.

```
public class FakeBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    private final int bankAccountNumber;
    private int balance;
    private int overdraft;

    public FakeBankAccountDatabaseConnection(int bankAccountNumber) {
        this.bankAccountNumber = bankAccountNumber;
    }

    @Override
    public int createBankAccount() {
        return bankAccountNumber;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return balance;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        this.balance = amount;
    }

    // ...
}
```

Fake

Fakes

Method
under
test

```
public void withdraw(int amount) {
    if (amount <= 0) {
        throw new BankAccountException("Cannot withdraw a zero or negative amount");
    }
    int balance = getBalance();
    int maxWithdrawalAmount = balance + getOverdraft();
    if (amount > maxWithdrawalAmount) {
        throw new BankAccountException(
            "Cannot withdraw more than the current balance plus the overdraft");
    }
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, balance - amount);
}
```

Test
using
fake

```
@Test
public void shouldCalculateBalanceCorrectlyFollowingAWithdrawal() {
    FakeBankAccountDatabaseConnection fake = new FakeBankAccountDatabaseConnection(0);

    // Given a bank account with a balance of £500 and an overdraft of £0
    BankAccount bankAccount = new BankAccount(fake, 500, 0);

    // When £100 is withdrawn
    bankAccount.withdraw(100);

    // Then the balance should be £400
    assertThat(bankAccount.getBalance(), equalTo(400));
}
```

Mocks

Mocks extend the idea of a stub – they allow you to control the values returned by a method but also can also **confirm methods were called with the correct values as arguments.**

Method
under
test

```
public void deposit(int amount) {  
    if (amount <= 0) {  
        throw new BankAccountException("Cannot deposit a zero or negative amount");  
    }  
    bankAccountDatabaseConnection.setBalance(bankAccountNumber, getBalance() + amount);  
}
```

Unless we use a fake (and write more tests for the fake), there is no way to ascertain that the value going into the database to set the bank account's balance is the correct one.

Mocks

Test using mock

```
@Test
public void shouldDepositAmount() {
    MockedBankAccountDatabaseConnection mock
        = new MockedBankAccountDatabaseConnection();

    // Given a Bank account with an opening balance of £100 and no overdraft
    BankAccount bankAccount = new BankAccount(mock, 100, 0);

    // When £100 is deposited
    bankAccount.deposit(100);

    // Then a call should be made to set the balance of the account to £200
    // (as signalled by a flag - mock.verify - being set to true)
    assertThat(mock.verify, equalTo(true));
}
```

Mock

```
public class MockedBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    public boolean verify = false;

    @Override
    public int createBankAccount() {
        return 1000;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 100;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        verify = (bankAccountNo == 1000 && amount == 200);
    }

    // ...
}
```

Explicitly verify the database is instructed to set the balance amount to £200 for the account no. 1000.

Spies

Spies are like mocks but without stubbed methods (methods that return predetermined values).

That is, they just do the **method call logging and checking** part.

They're useful for checking the interface between a unit and an external component. (Sometimes they're even used as part of integration tests.)

For example, they could be used to spy on methods and check that the correct SQL has been generated.

Or, that the contents of an email are as expected, before a service is invoked to send it.

Take Care With Doubles

Note how many of the examples involved a lot of implementation detail about the classes being doubled. In particular:

- **Fakes** need their own tests(!), since they involve more implementation
- **Mocks** record details about individual method calls, making them liable to brittleness.

As such, use doubles with care, and only when necessary.

Keeping things as real as possible is often the best way, and avoiding doubles altogether. (More on this to come.)

Mockito

Writing a new test doubles each time you want to test something can get quite painful, quite quickly.

Mockito is a useful framework for generating mocks for use with JUnit.

Since mocks are stubs and spies, and stubs are more specialised versions of dummies, Mockito can generate all types of double except fakes.

Mock example with Mockito

```
@Test
public void shouldDepositAmount() {
    MockedBankAccountDatabaseConnection mock
        = new MockedBankAccountDatabaseConnection();

    // Given a Bank account with an opening balance of £100 and no overdraft
    BankAccount bankAccount = new BankAccount(mock, 100, 0);

    // When £100 is deposited
    bankAccount.deposit(100);

    // Then a call should be made to set the balance of the account to £200
    // (as signalled by a flag - mock.verify - being set to true)
    assertThat(mock.verify, equalTo(true));
}
```

Test using manually
written mock

Manually- written
mock class

```
public class MockedBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    public boolean verify = false;

    @Override
    public int createBankAccount() {
        return 1000;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 100;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        verify = (bankAccountNo == 1000 && amount == 200);
    }

    // ...
}
```


Mock example with Mockito

```
@Test
public void shouldDepositAmount() {
    // Setup the mock, including to return a bank account number of 1000
    BankAccountDatabaseConnection mock = mock();
    when(mock.createBankAccount()).thenReturn(1000);

    // Given a Bank account with an opening balance of £100 and no overdraft
    BankAccount bankAccount = new BankAccount(mock, 100, 0);

    // Set the database mock to return a balance of £100 for this account number
    // as would have been instructed to have been set in the database via
    // the constructor
    when(mock.getBalance(1000)).thenReturn(100);

    // When £100 is deposited
    bankAccount.deposit(100);

    // Then a call should be made to set the balance of the account to £200
    verify(mock).setBalance(1000, 200);
}
```

Generate the mock object. We never (and don't need) to see any actual code – since it doesn't exist anyway

Generate “stubbed” methods for our mock

Verify that certain calls were made to the mock. Was `setBalance` called on it with a bank account no. of `1000`, with a balance of `200`?

Test using virtual mock

Fake Turned Into a Mock

```
@Test
public void shouldCalculateBalanceCorrectlyFollowingAWithdrawal() {
    FakeBankAccountDatabaseConnection fake = new FakeBankAccountDatabaseConnection(0);

    // Given a bank account with a balance of £500 and an overdraft of £0
    BankAccount bankAccount = new BankAccount(fake, 500, 0);

    // When £100 is withdrawn
    bankAccount.withdraw(100);

    // Then the balance should be £400
    assertThat(bankAccount.getBalance(), equalTo(400));
}
```

**Test using manually
written fake**

**Manually- written
fake class.**

It's just for testing, but
we're going to need to
test it as well!

```
public class FakeBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    private final int bankAccountNumber;
    private int balance;
    private int overdraft;

    public FakeBankAccountDatabaseConnection(int bankAccountNumber) {
        this.bankAccountNumber = bankAccountNumber;
    }

    @Override
    public int createBankAccount() {
        return bankAccountNumber;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return balance;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
        this.balance = amount;
    }

    // ...
}
```

Fake Turned Into a Mock

```
@Test
public void shouldCalculateBalanceCorrectlyFollowingAWithdrawal() {
    // Setup the mock, including to return a bank account number of 1000
    BankAccountDatabaseConnection mock = mock();
    when(mock.createBankAccount()).thenReturn(1000);

    // Given a bank account with a balance of £500 and an overdraft of £0
    BankAccount bankAccount = new BankAccount(mock, 500, 0);

    // Set the database mock to return a balance of £500 for this account number
    // as would have been instructed to have been set in the database via
    // the constructor
    when(mock.getBalance(1000)).thenReturn(500);

    // When £100 is withdrawn
    bankAccount.withdraw(100);

    // Then the balance should be £400
    verify(mock).setBalance(1000, 400);
}
```

We can just use a mock instead.

This code is similar to the last example of a mock.

Dummy Example with Mockito

```
@Test
public void shouldNotAllowNegativeAmountsToBeWithdrawn() {
    DummyBankAccountDatabaseConnection dummy = new DummyBankAccountDatabaseConnection();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(dummy, 0, 0);

    // When a negative amount is withdrawn, Then an exception is thrown
    assertThrows(BankAccountException.class, () -> {
        bankAccount.withdraw(-1000);
    });
}
```

**Test using manually
written dummy**

**Manually-written
dummy class**

```
public class DummyBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 0;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {
    }

    @Override
    public int getOverdraft(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setOverdraft(int bankAccountNo, int amount) {
    }
}
```


Dummy Example with Mockito

```
@Test
public void shouldNotAllowNegativeAmountsToBeWithdrawn() {
    // Setup the mock
    BankAccountDatabaseConnection mock = mock();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(mock, 0, 0);

    // When a negative amount is withdrawn, Then an exception is thrown
    assertThrows(BankAccountException.class, () -> {
        bankAccount.withdraw(-1000);
    });
}
```

Generate the mock object. Since we don't go and stub or verify any methods it's effectively a dummy

Test using virtual mock (which is a dummy in this case)

Stub Example with Mockito

```
@Test
public void shouldAssignABankAccountWhenOpeningAnAccount() {
    StubbedBankAccountDatabaseConnection stub = new StubbedBankAccountDatabaseConnection();

    // Given a bank account
    BankAccount bankAccount = new BankAccount(stub, 0, 0);

    // Then it should have a bank account number
    assertThat(bankAccount.getBankAccountNumber(), equalTo(1000));
}
```

Test using manually
written stub

Manually- written
stub class

```
public class StubbedBankAccountDatabaseConnection
    implements BankAccountDatabaseConnection {

    @Override
    public int createBankAccount() {
        return 1000;
    }

    @Override
    public int getBalance(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setBalance(int bankAccountNo, int amount) {

    }

    @Override
    public int getOverdraft(int bankAccountNo) {
        return 0;
    }

    @Override
    public void setOverdraft(int bankAccountNo, int amount) {

    }
}
```

Stub Example with Mockito

```
@Test
public void shouldAssignABankAccountWhenOpeningAnAccount() {
    // Setup the mock, including to return a bank account number of 1000
    BankAccountDatabaseConnection mock = mock();
    when(mock.createBankAccount()).thenReturn(1000);

    // Given a bank account
    BankAccount bankAccount = new BankAccount(mock, 0, 0);

    // Then it should have a bank account number
    assertThat(bankAccount.getBankAccountNumber(), equalTo(1000));
}
```

Generate the mock object and “stub” a method. Since we don’t go and verify any methods it’s effectively a stub.

Mockito – Summary

Mockito can save a lot of work in manually writing doubles.

Mockito can do more than we have covered here, see
<https://site.mockito.org/>

... but the temptation is to reach for it without considering alternatives.

Doubles can lead to brittle tests.

Always consider whether an integration test is more appropriate.