

CS608

Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

Notes on CS608 Exam

- All previous exam papers available on library website
 - <https://www.maynoothuniversity.ie/library/exam-papers>
- More recent ones are more representative of what to expect this year
 - 2024, 2023, 2022, 2021
- In particular, the test terminology has been updated from Roper's usage to the IEEE standard (as of Jan 2021)
 - Esp. "**test case**" and "**test data**"
- University supports available (next slide)

**Exams are coming up.
We know this can be a stressful time.
So, it's important to check-in and ask yourself...**

**How am
I feeling?**

**What tools do I
have to help me
cope?**

**Do I need some
extra support?**

**Scan now for resources
+ on-campus and off-campus supports:
Don't delay, act today!**

#BecomeYourOwnChampion



CS608

Test Automation

(Essentials of Software Testing, Chapter 11)

Overview

- Manual testing is slow, error-prone, and hard to repeat

Overview

- Manual testing is slow, error-prone, and hard to repeat
- Software testing needs to be fast, accurate, and repeatable

Overview

- Manual testing is slow, error-prone, and hard to repeat
- Software testing needs to be fast, accurate, and repeatable
- The solution is test automation

Overview

- Manual testing is slow, error-prone, and hard to repeat
- Software testing needs to be fast, accurate, and repeatable
- The solution is test automation
- Using test design techniques that have been discussed in previous chapters (e.g. EP, BVA, DT, SC, BC, OOT, User Stories, etc.)

Topics

1. Introduction
2. Test Frameworks: TestNG
3. Organising Automated Test Code
4. Setup and Cleanup Methods
5. In-line Tests vs Parameterised Tests
6. Test Coverage Measurement
7. Timeouts and Exceptions
8. Inheritance Testing
9. Testing private methods
10. Interfacing to Web Applications
11. Interfacing to Desktop Applications
12. Interfacing to Mobile Applications

1. Introduction

- Software testing needs to be **fast**, so that it can be performed frequently without a time penalty

1. Introduction

- Software testing needs to be **fast**, so that it can be performed frequently without a time penalty
- It needs to be **accurate** so that the test results can be relied on as a quality indicator

1. Introduction

- Software testing needs to be **fast**, so that it can be performed frequently without a time penalty
- It needs to be **accurate** so that the test results can be relied on as a quality indicator
- And it needs to be **repeatable** to allow for regression testing, where the same tests may be run many times for different software versions

1. Introduction

- Software testing needs to be **fast**, so that it can be performed frequently without a time penalty
- It needs to be **accurate** so that the test results can be relied on as a quality indicator
- And it needs to be **repeatable** to allow for regression testing, where the same tests may be run many times for different software versions
- This is particularly true for modern agile development approaches, where small increments of code are added and tested in a rapid cycle

Automation

- Some of the testing tasks that can be automated relatively easily are:
 - Execution of tests
 - Collection of test results
 - Evaluation of test results
 - Generation of test reports
 - Measurement of simple white-box test coverage

Automation

- Some of the testing tasks that can be automated relatively easily are:
 - Execution of tests
 - Collection of test results
 - Evaluation of test results
 - Generation of test reports
 - Measurement of simple white-box test coverage
- Some of the testing tasks that are more difficult to automate are:
 - Generation of test conditions/test coverage items and the data for test cases
 - Measurement of black-box test coverage
 - Measurement of complex white-box test coverage

What is Automated

- Rule-of-thumb: if a test is to be executed more than twice, automate it

What is Automated

- Rule-of-thumb: if a test is to be executed more than twice, automate it
- Unit test execution is invariably automated:
 - Implemented by code that calls the required methods
 - With the specified test input data
 - And compares the actual results with the expected results

What is Automated

- Rule-of-thumb: if a test is to be executed more than twice, automate it
- Unit test execution is invariably automated:
- Application tests are more difficult to automate
 - In manual testing, correctness of output on the screen left to the tester's judgement
 - Automated testing requires knowing details of how expected results are displayed
 - Application tests more complex to automate. Depend on the details of the system interface: inputs are provided and results collected via the system interface
 - It generally takes extra time to develop automated application tests
 - Extra complex program interface library to be used (e.g. Selenium)
 - In general there is a shift from manual to automated application testing

Grouping and Reports

- Automated tests can be grouped into collections (which are also referred to as *test sets* or *test suites*)
- And the results automatically collated into a report
 - Overall summary of the test result (pass or fail)
 - Statistics on the tests that have been run
 - Test incident report on each failure, providing exact details on why the test failed in order to assist in locating the fault.

Tools

- Automated unit testing examined in more detail, using TestNG as an **example** test framework
- Automated **application** testing examined in more detail, using TestNG to manage the tests, Selenium to interface tests with web-based applications
- Note: just examples, representative of typical test automation tool **features**

Interpreting Test Results

- Automated testing may generate a number of different test results:
 - PASSED – the test has passed.
 - FAILED – the test has failed.
 - SKIPPED – the test was not run (for example if some setup failed to execute properly prior to the test)
 - NOT SELECTED – the test was in the test collection, but deselected for execution

Interpreting Test Results

- Automated testing may generate a number of different test results:
 - PASSED – the test has passed.
 - FAILED – the test has failed.
 - SKIPPED – the test was not run (for example if some setup failed to execute properly prior to the test)
 - NOT SELECTED – the test was in the test collection, but deselected for execution
- Differentiate between tests that failed, and tests that were skipped
- Tests are usually skipped because the test setup failed: perhaps the class file for the code being tested was not in the expected location, or the web driver was the wrong version for the web browser in use
- Skipped tests generally require a response by the tester to rerun them

Documenting Automated Tests

- Test documentation:
 - analysis results, test coverage items, test cases/data
- The test case identifiers should be referenced in the test method names, or in the data fields for parameterised tests
- When automating software tests, there are two choices for where to store the test documentation:
 1. In separate files (perhaps using word processing and spreadsheet tools, or a database in a larger organisation)
 2. To store it as comments in the test programs

Tests and Files

- Note that all the tests within a file must have unique identifiers (in this module they are numbered using a simple hierarchy)
- Each test can then be uniquely identified using the test item and the test identifier: for example giveDiscount() test T1.3
- This allows a specific test to be rerun in order to replicate the failure as part of the debugging process, and also for it to be re-executed to verify that a fault has been correctly fixed
- It is useful to maintain an edit history of the test file (e.g. GIT)
- Put all tests into one file, or multiple files (e.g. BBT and WBT)
- Test runners support multiple files (e.g. TestNG)

Software Test Automation & Version Control

- Test reports must include an identifier that uniquely states the software being tested (its name, version, variant, etc.)
- This means that the report can be interpreted in the correct context, and any debugging and fixes applied to the correct version (or variant) of the software
- It also enables the verification that the fixes have been made to that specific version (or variant)
- For a class, the identifier is generally the version number from the version control system
- For complete systems, this is generally the build number from the build procedure (or it may be the release tag, or the date for revision control systems similar to Git)

2. Test Frameworks: TestNG

- TestNG is used as a representative unit test automation framework
- Other frameworks have similar features: e.g. Cucumber/Gherkin
- Using TestNG to introduce and explain typical features required for automated testing

2. Test Frameworks: TestNG

- Only a limited subset of the TestNG features are described
- TestNG consists of a set of Java classes that automate the execution of software tests, collect and evaluate the test results, and generate test reports
- TestNG tests are executing using a *test runner*

A Detailed Look at a TestNG Example

- Import required TestNG libraries
- @Test annotation for tests
- Use assertions in tests:
 - Assertion fails, test fails and halts
 - Otherwise, test passes
- Class Demo has the methods:
 - setValue(int) – setter for 'value'
 - getValue() – getter for 'value'
 - add(int) – add to 'value'

```
1  package example;
2
3  import static org.testng.Assert.*;
4  import org.testng.annotations.*;
5
6  public class DemoTest {
7
8      @Test
9      public void test1() {
10         Demo d = new Demo();
11         d.setValue(56);
12         d.add(44);
13         assertEquals( d.getValue(), 100 );
14     }
15
16 }
```

A Detailed Look at a TestNG Example

- Line 10 – object to be tested is created

```
1  package example;
2
3  import static org.testng.Assert.*;
4  import org.testng.annotations.*;
5
6  public class DemoTest {
7
8      @Test
9      public void test1() {
10         Demo d = new Demo();
11         d.setValue(56);
12         d.add(44);
13         assertEquals( d.getValue(), 100 );
14     }
15
16 }
```

A Detailed Look at a TestNG Example

- Line 10 – object to be tested is created
- Line 11 – object is initialised (put into correct state for testing)
 - value is set to 56

```
1  package example;
2
3  import static org.testng.Assert.*;
4  import org.testng.annotations.*;
5
6  public class DemoTest {
7
8      @Test
9      public void test1() {
10         Demo d = new Demo();
11         d.setValue(56);
12         d.add(44);
13         assertEquals( d.getValue(), 100 );
14     }
15
16 }
```

A Detailed Look at a TestNG Example

- Line 10 – object to be tested is created
- Line 11 – object is initialised (put into correct state for testing)
- Line 12 – method under test is called
 - add is called with the input test data 44

```
1  package example;
2
3  import static org.testng.Assert.*;
4  import org.testng.annotations.*;
5
6  public class DemoTest {
7
8      @Test
9      public void test1() {
10         Demo d = new Demo();
11         d.setValue(56);
12         d.add(44);
13         assertEquals( d.getValue(), 100 );
14     }
15
16 }
```

A Detailed Look at a TestNG Example

- Line 10 – object to be tested is created
- Line 11 – object is initialised (put into correct state for testing)
- Line 12 – method under test is called
- Line 13 – output data is collected
 - `getValue()`

```
1  package example;
2
3  import static org.testng.Assert.*;
4  import org.testng.annotations.*;
5
6  public class DemoTest {
7
8      @Test
9      public void test1() {
10         Demo d = new Demo();
11         d.setValue(56);
12         d.add(44);
13         assertEquals( d.getValue(), 100 );
14     }
15
16 }
```


A Detailed Look at a TestNG Example

- Line 10 – object to be tested is created
- Line 11 – object is initialised (put into correct state for testing)
- Line 12 – method under test is called
- Line 13 – output data is collected
- Line 13 – actual results compared with expected results
 - assertEquals()
 - Expected results: return value 100

```
1  package example;
2
3  import static org.testng.Assert.*;
4  import org.testng.annotations.*;
5
6  public class DemoTest {
7
8      @Test
9      public void test1() {
10         Demo d = new Demo();
11         d.setValue(56);
12         d.add(44);
13         assertEquals(d.getValue(), 100);
14     }
15
16 }
```

TestNG Test Output

- Test result for each test method run
- The test result for each set of tests run together
 - *Command line test* if run from command line
 - Number of tests run, failed, skipped
- A summary of the entire test suite
 - Multiple sets of tests can be collected into a suite
 - Default *command line suite* used here

```
PASSED: test1
```

```
=====
```

```
Command line test
```

```
Tests run: 1, Failures: 0, Skips: 0
```

```
=====
```

```
Command line suite
```

```
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
```

```
=====
```

TestNG Testing

- TestNG supports a number of assertion methods – the most commonly used ones are:
 - assertEquals(), assertTrue(), and assertFalse()
- If assertion passes, execution continues to the next line in the method
- If assertion fails, test method terminates with an exception, causing the test to fail
- This is why it is not a good idea to have multiple tests in a single test method: if one fails, then the subsequent tests are not run – this does not apply to parameterised tests though

TestNG Test Runner

- The test methods are identified by a TestNG *Test Runner* which:
 - Uses Java Reflection to find all methods in test class with @Test annotation
 - Calls these in turn
 - Trapping exceptions, and keeping counters for the numbers of tests run, tests passed, and tests failed
- Running tests in TestNG:
 - Default command-line test runner
 - Run within an IDE (such as Eclipse)
 - Managed using an XML file, which is passed to the test runner

3. Organising Automated Test Code

- Recommendation: have a test class for every program class
- E.g. class Demo in Demo.java has a test class DemoTest in DemoTest.java
- You can use any naming convention you like; I use the convention of adding the word *Test* after the class name
- The way to group multiple tests together is to put one test per method, and group the test methods into suites of tests, sometimes referred to as *test sets*
- To increase flexibility, test suites may themselves be grouped into larger test suites
- Note: parameterised tests contain more than one test case per test method

Organising TestNG Tests with an XML file

- An XML file can be passed as a parameter to the TestNG runner
- This can organise the test methods into suites and tests with test classes and test methods
- Note: TestNG terminology is not fully aligned with the IEEE terminology used elsewhere...

XML Example

- One suite: "Suite1"
- Two (groups of) tests:
 - "standardTest"
 - "extraTest"
- **standardTest** uses
 - Class: Example.DemoTest
 - Method: test1
- **extraTest** uses
 - Class: Example.DemoTestExtra
 - Methods: test2, test3



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Suite1">
  <test name="standardTest">
    <classes>
      <class name="example.DemoTest">
        <methods>
          <include name="test1" />
        </methods>
      </class>
    </classes>
  </test>
  <test name="extraTest">
    <classes>
      <class name="example.DemoTestExtra">
        <methods>
          <include name="test2" />
          <include name="test3" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

XML Example

- One suite: "Suite1"
- Two (groups of) tests:
 - "standardTest"
 - "extraTest"
- **standardTest** uses
 - Class: Example.DemoTest
 - Method: test1
- **extraTest** uses
 - Class: Example.DemoTestExtra
 - Methods: test2, test3

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

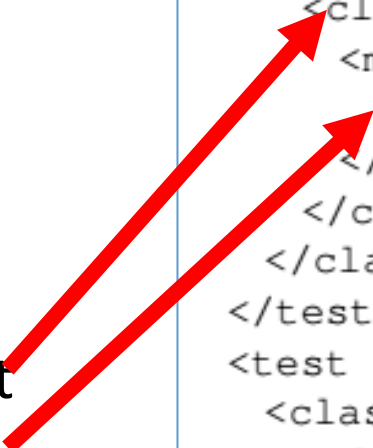
<suite name="Suite1">
  <test name="standardTest">
    <classes>
      <class name="example.DemoTest">
        <methods>
          <include name="test1" />
        </methods>
      </class>
    </classes>
  </test>
  <test name="extraTest">
    <classes>
      <class name="example.DemoTestExtra">
        <methods>
          <include name="test2" />
          <include name="test3" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```


XML Example

- One suite: "Suite1"
- Two (groups of) tests:
 - "standardTest"
 - "extraTest"
- **standardTest** uses
 - Class: Example.DemoTest
 - Method: test1
- **extraTest** uses
 - Class: Example.DemoTestExtra
 - Methods: test2, test3

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Suite1">
  <test name="standardTest">
    <classes>
      <class name="example.DemoTest">
        <methods>
          <include name="test1" />
        </methods>
      </class>
    </classes>
  </test>
  <test name="extraTest">
    <classes>
      <class name="example.DemoTestExtra">
        <methods>
          <include name="test2" />
          <include name="test3" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```



XML Example

- One suite: "Suite1"
- Two (groups of) tests:
 - "standardTest"
 - "extraTest"
- **standardTest** uses
 - Class: Example.DemoTest
 - Method: test1
- **extraTest** uses
 - Class: Example.DemoTestExtra
 - Methods: test2, test3

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Suite1">
  <test name="standardTest">
    <classes>
      <class name="example.DemoTest">
        <methods>
          <include name="test1" />
        </methods>
      </class>
    </classes>
  </test>
  <test name="extraTest">
    <classes>
      <class name="example.DemoTestExtra">
        <methods>
          <include name="test2" />
          <include name="test3" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```



Output using TestNG with XML File

- TestNG parameter 'log level' 3 used here:
 - Specifies the detail (1-5)
- Test methods called & results
- Results at each level of the hierarchy as defined in the XML file
- Using the XML file as a basis, the tester can then select particular tests, test classes, or test suites to execute

```
===== Invoked methods
    DemoTest.test1()[pri:0, instance:example.DemoTest@6e171cd7] 1847008471
=====
PASSED: test1

=====
    standardTest
    Tests run: 1, Failures: 0, Skips: 0
=====
PASSED: test1

=====
    extraTest
    Tests run: 1, Failures: 0, Skips: 0
=====

===== Invoked methods
    DemoTestExtra.test2()[pri:0, instance:example.DemoTestExtra@4f6ee6e4]
    1332668132
    DemoTestExtra.test3()[pri:0, instance:example.DemoTestExtra@4f6ee6e4]
    1332668132
=====
```

4. Setup and Cleanup Methods

- Test automation tools typically support ways to run particular additional methods before and after every test class, or before and after every test method (or collections of test methods) in a test class
- This allows objects (or connections to external software, such as servers or databases) to be setup before a test, and cleaned up afterwards
- For example, if an object needs to be shared between all the tests in a class, it can be created before all the tests are run, and re-initialised before every individual test

Example: @BeforeMethod

- Setup() creates a new instance of Demo
- The test methods test1 and test2 each execute against individual instances of Demo
- setup() is executed before every test method

```
6 public class DemoTestM {
7
8     public Demo d;
9
10    @BeforeMethod public void setup() {
11        System.out.println("Creating a new Demo object");
12        d = new Demo();
13    }
14
15    @Test
16    public void test1() {
17        d.setValue(100);
18        assertEquals( d.getValue(), 100 );
19    }
20
21    @Test
22    public void test2() {
23        d.setValue(200);
24        assertEquals( d.getValue(), 200 );
25    }
26
27 }
```

Output with @BeforeMethod

- A new Demo object is created for each test method
- Output lines appear out of order:
 - TestNG test results all reported at the end of the complete test, not after each test method
 - println() output is shown immediately

```
Creating a new Demo object
Creating a new Demo object
PASSED: test1
PASSED: test2
=====
Command line suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
=====
```

@BeforeClass and @AfterClass

- To use same object every test, a single object can be created before any of the tests are run using the @BeforeClass annotation
- Objects can be cleaned up after all the tests have run using the @AfterClass annotation
- By annotating setup() and cleanup() as shown in the example, test1 and test2 would each execute against the same instance of Demo

```
1  @BeforeClass public void setup() {  
2      d = new Demo();  
3  }  
4  
5  @AfterClass public void cleanup() {  
6      d = null;  
7  }
```

Commonly Used Annotations

- For Suites/Classes/Groups/Methods:
- `@BeforeSuite/@AfterSuite`
 - called before/after a defined suite of tests
- `@BeforeClass/@AfterClass`
 - called before the execution of the first test method in a test class / after the execution of the last test method in a test class
- `@BeforeGroups/@AfterGroups`
 - called before/after a defined group of tests
- `@BeforeMethod/@AfterMethod`
 - called before/after the execution of every test method

5. In-line Tests vs Parameterised Tests

- All test frameworks support the ability to define test methods with in-line test data (included as constants inside the method)
- In TestNG, the `@Test` annotation is used
- Note: `test2()` and `test3()` are slightly different
- Output shows each test & the result – but no data

```
public class DemoTestInline {  
    @Test public void test1() {  
        Demo d = new Demo();  
        d.setValue(56);  
        d.add(44);  
        assertEquals( d.getValue(), 100 );  
    }  
    @Test  
    public void test2() {  
        Demo d = new Demo();  
        d.setValue(0);  
        assertEquals( d.getValue(), 0 );  
    }  
    @Test  
    public void test3() {  
        Demo d = new Demo();  
        d.setValue(-1000);  
        d.add(-1234);  
        assertEquals( d.getValue(), -2234 );  
    }  
}
```

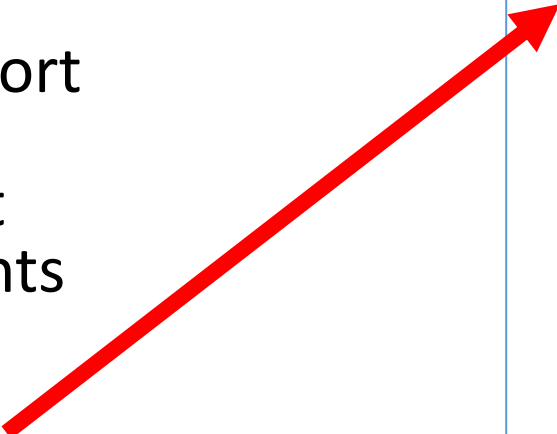
```
PASSED: test1  
PASSED: test2  
PASSED: test3  
=====
```

Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

5. In-line Tests vs Parameterised Tests

- All test frameworks support the ability to define test methods with in-line test data (included as constants inside the method)
- In TestNG, the `@Test` annotation is used
- Note: `test2()` and `test3()` are slightly different
- Output shows each test & the result – but no data



```
public class DemoTestInline {  
    @Test public void test1() {  
        Demo d = new Demo();  
        d.setValue(56);  
        d.add(44);  
        assertEquals( d.getValue(), 100 );  
    }  
    @Test  
    public void test2() {  
        Demo d = new Demo();  
        d.setValue(0);  
        assertEquals( d.getValue(), 0 );  
    }  
    @Test  
    public void test3() {  
        Demo d = new Demo();  
        d.setValue(-1000);  
        d.add(-1234);  
        assertEquals( d.getValue(), -2234 );  
    }  
}
```

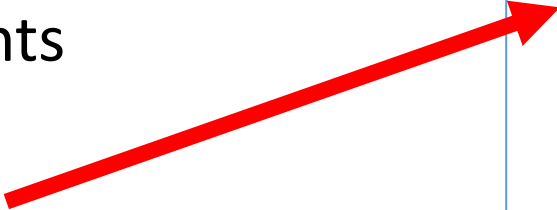
```
PASSED: test1  
PASSED: test2  
PASSED: test3  
=====
```

Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

5. In-line Tests vs Parameterised Tests

- All test frameworks support the ability to define test methods with in-line test data (included as constants inside the method)
- In TestNG, the `@Test` annotation is used
- Note: `test2()` and `test3()` are slightly different
- Output shows each test & the result – but no data



```
public class DemoTestInline {  
    @Test public void test1() {  
        Demo d = new Demo();  
        d.setValue(56);  
        d.add(44);  
        assertEquals( d.getValue(), 100 );  
    }  
    @Test  
    public void test2() {  
        Demo d = new Demo();  
        d.setValue(0);  
        assertEquals( d.getValue(), 0 );  
    }  
    @Test  
    public void test3() {  
        Demo d = new Demo();  
        d.setValue(-1000);  
        d.add(-1234);  
        assertEquals( d.getValue(), -2234 );  
    }  
}
```

```
PASSED: test1  
PASSED: test2  
PASSED: test3  
=====
```

Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

5. In-line Tests vs Parameterised Tests

- All test frameworks support the ability to define test methods with in-line test data (included as constants inside the method)
- In TestNG, the `@Test` annotation is used
- Note: `test2()` and `test3()` are slightly different
- Output shows each test & the result – but no data

```
public class DemoTestInline {  
    @Test public void test1() {  
        Demo d = new Demo();  
        d.setValue(56);  
        d.add(44);  
        assertEquals( d.getValue(), 100 );  
    }  
    @Test  
    public void test2() {  
        Demo d = new Demo();  
        d.setValue(0);  
        assertEquals( d.getValue(), 0 );  
    }  
    @Test  
    public void test3() {  
        Demo d = new Demo();  
        d.setValue(-1000);  
        d.add(-1234);  
        assertEquals( d.getValue(), -2234 );  
    }  
}
```

```
PASSED: test1  
PASSED: test2  
PASSED: test3  
=====
```

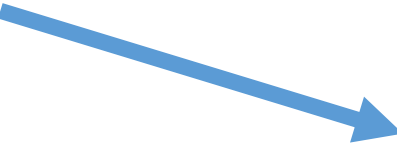
Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

5. In-line Tests vs Parameterised Tests

- All test frameworks support the ability to define test methods with in-line test data (included as constants inside the method)
- In TestNG, the `@Test` annotation is used
- Note: `test2()` and `test3()` are slightly different
- Output shows each test & the result – but no data

```
public class DemoTestInline {  
    @Test public void test1() {  
        Demo d = new Demo();  
        d.setValue(56);  
        d.add(44);  
        assertEquals( d.getValue(), 100 );  
    }  
    @Test  
    public void test2() {  
        Demo d = new Demo();  
        d.setValue(0);  
        assertEquals( d.getValue(), 0 );  
    }  
    @Test  
    public void test3() {  
        Demo d = new Demo();  
        d.setValue(-1000);  
        d.add(-1234);  
        assertEquals( d.getValue(), -2234 );  
    }  
}
```



```
PASSED: test1  
PASSED: test2  
PASSED: test3  
=====
```

Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

Parameterised Tests (TestNG DataProvider)

- The method *test()* is called sequentially with each row of test data in order as follows:
- `test("test1", 56, 44, 100);`
- `test("test2", 0, 0, 0);`
- `test("test3", -1000, -1234, -2234);`
- Slight difference in output – the name of the parameterised test method and the values of the parameters are shown for each test executed

```
public class DemoTestParam {  
    private static Object[][] testData = new Object[][] {  
        { "test1", 56, 44, 100 },  
        { "test2", 0, 0, 0 },  
        { "test3", -1000, -1234, -2234 },  
    };  
    @DataProvider(name="testset1")  
    public Object[][] getTestData() {  
        return testData;  
    }  
    @Test(dataProvider="testset1")  
    public void test(String id, int x, int y, int er) {  
        Demo d = new Demo();  
        d.setValue(x);  
        d.add(y);  
        assertEquals( d.getValue(), er );  
    }  
}
```

```
PASSED: test("test1", 56, 44, 100)  
PASSED: test("test2", 0, 0, 0)  
PASSED: test("test3", -1000, -1234, -2234)  
=====
```

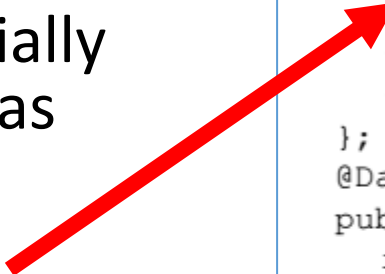
Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

Parameterised Tests (TestNG DataProvider)

- The method *test()* is called sequentially with each row of test data in order as follows:
- `test("test1", 56, 44, 100);`
- `test("test2", 0, 0, 0);`
- `test("test3", -1000, -1234, -2234);`
- Slight difference in output – the name of the parameterised test method and the values of the parameters are shown for each test executed

```
public class DemoTestParam {  
    private static Object[][] testData = new Object[][] {  
        { "test1", 56, 44, 100 },  
        { "test2", 0, 0, 0 },  
        { "test3", -1000, -1234, -2234 },  
    };  
    @DataProvider(name="testset1")  
    public Object[][] getTestData() {  
        return testData;  
    }  
    @Test(dataProvider="testset1")  
    public void test(String id, int x, int y, int er) {  
        Demo d = new Demo();  
        d.setValue(x);  
        d.add(y);  
        assertEquals( d.getValue(), er );  
    }  
}
```



```
PASSED: test("test1", 56, 44, 100)  
PASSED: test("test2", 0, 0, 0)  
PASSED: test("test3", -1000, -1234, -2234)  
=====
```

Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

Parameterised Tests (TestNG DataProvider)

- The method *test()* is called sequentially with each row of test data in order as follows:
- `test("test1", 56, 44, 100) :`
- `test("test2", 0, 0, 0) ;`
- `test("test3", -1000, -1234, -2234) ;`
- Slight difference in output – the name of the parameterised test method and the values of the parameters are shown for each test executed

```
public class DemoTestParam {  
    private static Object[][] testData = new Object[][] {  
        { "test1", 56, 44, 100 },  
        { "test2", 0, 0, 0 },  
        { "test3", -1000, -1234, -2234 },  
    };  
    @DataProvider(name="testset1")  
    public Object[][] getTestData() {  
        return testData;  
    }  
    @Test(dataProvider="testset1")  
    public void test(String id, int x, int y, int er) {  
        Demo d = new Demo();  
        d.setValue(x);  
        d.add(y);  
        assertEquals( d.getValue(), er );  
    }  
}
```

```
PASSED: test("test1", 56, 44, 100)  
PASSED: test("test2", 0, 0, 0)  
PASSED: test("test3", -1000, -1234, -2234)  
=====
```

Command line suite

```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```


Parameterised Tests (TestNG DataProvider)

- The method *test()* is called sequentially with each row of test data in order as follows:
- `test("test1", 56, 44, 100);`
- `test("test2", 0, 0, 0);`
- `test("test3", -1000, -1234, -2234);`
- Slight difference in output – the name of the parameterised test method and the values of the parameters are shown for each test executed

```
public class DemoTestParam {  
    private static Object[][] testData = new Object[][] {  
        { "test1", 56, 44, 100 },  
        { "test2", 0, 0, 0 },  
        { "test3", -1000, -1234, -2234 },  
    };  
    @DataProvider(name="testset1")  
    public Object[][] getTestData() {  
        return testData;  
    }  
    @Test(dataProvider="testset1")  
    public void test(String id, int x, int y, int er) {  
        Demo d = new Demo();  
        d.setValue(x);  
        d.add(y);  
        assertEquals( d.getValue(), er );  
    }  
}
```

```
PASSED: test("test1", 56, 44, 100)  
PASSED: test("test2", 0, 0, 0)  
PASSED: test("test3", -1000, -1234, -2234)  
=====
```

Command line suite


```
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

Parameterised Tests (TestNG DataProvider)

- The method *test()* is called sequentially with each row of test data in order as follows:
- `test("test1", 56, 44, 100);`
- `test("test2", 0, 0, 0);`
- `test("test3", -1000, -1234, -2234);`
- Slight difference in output – the name of the parameterised test method and the values of the parameters are shown for each test executed

```
public class DemoTestParam {  
    private static Object[][] testData = new Object[][] {  
        { "test1", 56, 44, 100 },  
        { "test2", 0, 0, 0 },  
        { "test3", -1000, -1234, -2234 },  
    };  
    @DataProvider(name="testset1")  
    public Object[][] getTestData() {  
        return testData;  
    }  
    @Test(dataProvider="testset1")  
    public void test(String id, int x, int y, int er) {  
        Demo d = new Demo();  
        d.setValue(x);  
        d.add(y);  
        assertEquals( d.getValue(), er );  
    }  
}
```

```
PASSED: test("test1", 56, 44, 100)  
PASSED: test("test2", 0, 0, 0)  
PASSED: test("test3", -1000, -1234, -2234)  
=====
```



```
Command line suite  
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0  
=====
```

Using Iterators

- TestNG supports both **static** and **dynamic** data providers

Using Iterators

- TestNG supports both static and dynamic data providers
- **Static** data providers return a fixed array (or an Iterator over a standard collection)
 - We have used this already in our parameterised tests

Using Iterators

- TestNG supports both static and dynamic data providers
- Static data providers return a fixed array (or an Iterator over a standard collection)
- **Dynamic** data providers return a customised Iterator, which can generate data on-the-fly
- **Dynamic** data providers can be written to provide test data which changes based on the test progress, or to support very large data sets

Example Iterator

```
private static Object[][] testData = new Object[][] {
    { "test1", 56, 44, 100 },
    { "test2", 0, 0, 0 },
    { "test3", -1000, -1234, -2234 },
};

@DataProvider(name="testset1")
public Iterator<Object[]> getData() {
    return new DataGenerator(testData);
}

@Test(dataProvider="testset1")
public void test(String id, int x, int y, int er) {
    Demo d = new Demo();
    d.setValue(x);
    d.add(y);
    assertEquals( d.getValue(), er );
}
```

```
static class DataGenerator implements Iterator<Object[]> {

    private int index=0;
    private Object[][] data;

    DataGenerator(Object[][] testData) {
        data = testData;
    }

    @Override public boolean hasNext() {
        return index<data.length;
    }

    @Override public Object[] next() {
        if (index<data.length)
            return data[index++];
        else
            return null;
    }
}
```

- Each time the iterator next() method is called, it returns the next row of data

Example Iterator

```
private static Object[][] testData = new Object[][] {
    { "test1", 56, 44, 100 },
    { "test2", 0, 0, 0 },
    { "test3", -1000, -1234, -2234 },
};

@DataProvider(name="testset1")
public Iterator<Object[]> getData() {
    return new DataGenerator(testData);
}

@Test(dataProvider="testset1")
public void test(String id, int x, int y, int er) {
    Demo d = new Demo();
    d.setValue(x);
    d.add(y);
    assertEquals( d.getValue(), er );
}
```

```
static class DataGenerator implements Iterator<Object[]> {

    private int index=0;
    private Object[][] data;

    DataGenerator(Object[][] testData) {
        data = testData;
    }

    @Override public boolean hasNext() {
        return index<data.length;
    }

    @Override public Object[] next() {
        if (index<data.length)
            return data[index++];
        else
            return null;
    }
}
```

- Each time the iterator next() method is called, it returns the next row of data
- Here, array used, but data can be dynamically generated within the next method

Example Iterator

```
private static Object[][] testData = new Object[][] {
    { "test1", 56, 44, 100 },
    { "test2", 0, 0, 0 },
    { "test3", -1000, -1234, -2234 },
};

@DataProvider(name="testset1")
public Iterator<Object[]> getData() {
    return new DataGenerator(testData);
}

@Test(dataProvider="testset1")
public void test(String id, int x, int y, int er) {
    Demo d = new Demo();
    d.setValue(x);
    d.add(y);
    assertEquals( d.getValue(), er );
}
```

```
static class DataGenerator implements Iterator<Object[]> {

    private int index=0;
    private Object[][] data;

    DataGenerator(Object[][] testData) {
        data = testData;
    }

    @Override public boolean hasNext() {
        return index<data.length;
    }

    @Override public Object[] next() {
        if (index<data.length)
            return data[index++];
        else
            return null;
    }
}
```

- Each time the iterator next() method is called, it returns the next row of data
- Here, array used, but data can be dynamically generated within the next method
- Particularly useful for random testing where data may be generated on demand

6. Test Coverage Measurement

- Most languages provide automated tools to measure at least statement coverage and branch coverage
- These tools can be used to verify that a white-box test actually achieves the coverage it is intended to achieve, and are mainly used in practice to measure the white-box coverage of black-box tests
- If only low coverage is achieved, then there are untested components in the code, and achieving higher levels of coverage requires the use of white-box testing techniques
- Code coverage can be measured for any type of testing.
- For Java there are a number of options: we use JaCoCo as an example

JaCoCo Example

- Lines 1-4, 6, 8-23, 26, 29-30, 35-36, 39, 41-43 have no source code
 - Not highlighted
- Lines 7, 24-25, 27-28, 31-34, 40 are fully executed
 - Green
- Line 36 is partially executed
 - Yellow
- Line 38 is not executed
 - Red
 - Also line 5: the constructor

JaCoCo Coverage Report > example > OnlineSales.java

OnlineSales.java

```
1. package example;
2. // Note: this version contains Fault 4
3. import static example.OnlineSales.Status.*;
4.
5. public class OnlineSales {
6.
7.     public static enum Status { FULLPRICE, DISCOUNT, ERROR };
8.
9.     /**
10.      * Determine whether to give a discount for online sales.
11.      * Gold customers get a discount above 80 bonus points.
12.      * Other customers get a discount above 120 bonus points.
13.      *
14.      * @param bonusPoints How many bonus points the customer has accumulated
15.      * @param goldCustomer Whether the customer is a Gold Customer
16.      *
17.      * @return
18.      * DISCOUNT - give a discount<br>
19.      * FULLPRICE - charge the full price<br>
20.      * ERROR - invalid inputs
21.      */
22.     public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23.     {
24.         Status rv = FULLPRICE;
25.         long threshold=120;
26.
27.         if (bonusPoints<=0)
28.             rv = ERROR;
29.
30.         else {
31.             if (goldCustomer)
32.                 threshold = 80;
33.             if (bonusPoints>threshold)
34.                 rv=DISCOUNT;
35.         }
36.         if (bonusPoints==43) // fault 4
37.             rv = DISCOUNT;
38.
39.         return rv;
40.     }
41. }
42.
43. }
```

Information on Branch Coverage

```
21.  */
22.  public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23.  {
24.      Status rv = FULLPRICE;
25.      long threshold=120;
26.
27.      ◆ if (bonusPoints<=0)
28.          rv = ERROR;
29.      All 2 branches covered.
30.      else {
31.          ◆ if (goldCustomer)
32.              threshold = 80;
33.          ◆ if (bonusPoints>threshold)
34.              rv=DISCOUNT;
35.      }
36.
37.      ◆ if (bonusPoints==43) // fault 4
38.          rv = DISCOUNT;
39.
40.      return rv;
41.  }
42.
43. }
```

- Hover on diamonds:
 - Line 27 “All 2 branches covered”
 - Line 31 “All 2 branches covered”
 - Line 33 “All 2 branches covered”
 - Line 37 “1 of 2 branches missed”

Coverage Summary

JaCoCo Coverage Report > example > OnlineSales

OnlineSales

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
OnlineSales()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
giveDiscount(long_boolean)	<div><div></div></div>	93%	<div><div></div></div>	87%	1	5	1	11	0	1
Total	5 of 32	84%	1 of 8	87%	2	6	2	12	1	2

Created with JaCoCo 0.8.5

- The key figures here are the missed instructions and missed branches for the method being tested: giveDiscount()
- This shows that:
 - 5 out of 32 executable instructions have not been executed
 - 1 of the 8 branches have not been taken
- Note that there are two completely different issues here: one is whether the tests have passed or not, and the second is whether full coverage has been achieved.

Lazy Evaluation

- Interpreting the statement coverage results when a line of source code is only partially executed can require some thought
- An optimisation feature called lazy evaluation may cause only some of the boolean conditions in a complex decision to be executed
- In Java, the Conditional-Or (||) and Conditional-And (&&) operators are guaranteed to be evaluated left-to-right; the right-hand operand is only evaluated if necessary

7 Timeouts and Exceptions

- Timeouts
 - Tests can timeout if they take too long to execute
- Some code throws exceptions
 - Must be able to identify if a test failure exception or an expected exception
 - Must be able to pass a test if **do** get an expected exception
 - Must be able to fail a test if **don't** get an expected exception

7(a). Timeouts

- Good practice to ensure tests do not continue to run for too long:
 - infinite loop
 - deadlock
- Less relevant for unit testing; critical for application testing
- In TestNG, the **timeOut** parameter is used

```
public class InfiniteTest {  
  
    @Test(timeOut=1000)  
    public void test1() {  
        assertEquals( Infinite.mul2(10), 20 );  
    }  
  
}
```

- test1() fails if the test takes more than 1000 milliseconds to execute

Example

- Faulty code with an infinite loop

```
class Infinite {  
    // return x*2  
    public static int mul2(int x) {  
        while (x>0)  
            x = (x * 2) - x;  
        return x;  
    }  
}
```

- Running the test with a timeout against this fault code results in a test failure, as shown

```
FAILED: test1  
org.testng.internal.thread.ThreadTimeoutException: Method example.InfiniteTest.  
    test1() didn't finish within the time-out 1000  
=====  
Command line suite  
Total tests run: 1, Passes: 0, Failures: 1, Skips: 0  
=====
```

- If no timeout was included, the test would not terminate.

7(b). Exceptions

- Exceptions are one of the principle ways for methods to indicate a failure/return an error in Java
- Most Java test frameworks use exceptions to report a test failure
- Therefore, if a method being tested is expected to raise an exception, this must be handled differently
 - to prevent the test failing incorrectly
 - also to verify that the exception is correctly raised
- In TestNG, the **expectedExceptions** parameter is used

Example

- add(x) throws an exception if x is not greater than 0
- The test must verify that
 - (a) an exception is not raised when x is valid
 - (b) an exception is raised when x is invalid

```
// Only add values greater than 0
public void add(int x) throws IllegalArgumentException
{
    if (x<1) throw new IllegalArgumentException("Invalid x");
    value += x;
}
```

```
@Test
public void test1() {
    d.setValue(0);
    d.add(44);
    assertEquals( d.getValue(), 44 );
}

@Test(expectedExceptions=IllegalArgumentException.class)
public void test2() {
    d.setValue(0);
    d.add(-44);
    assertEquals( d.getValue(), 44 );
}
```

Example

- When an unexpected exception is raised, such as when an assertion fails, then the test fails
- To notify TestNG that an exception is **expected**, parameters to the `@Test` annotation are used

```
// Only add values greater than 0
public void add(int x) throws IllegalArgumentException
{
    if (x<1) throw new IllegalArgumentException("Invalid x");
    value += x;
}
```

```
@Test
public void test1() {
    d.setValue(0);
    d.add(44);
    assertEquals( d.getValue(), 44 );
}
```

```
@Test(expectedExceptions=IllegalArgumentException.class)
public void test2() {
    d.setValue(0);
    d.add(-44);
    assertEquals( d.getValue(), 44 );
}
```

Test Results (Exceptions)

```
PASSED: test1
PASSED: test2
=====
Command line suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
=====
```

- The tests both pass
- If test1() had raised an exception then the test would have failed
- If test2() had not raised an exception, then that test would have failed

8. Inheritance Testing

- A subclass inherits “responsibilities” of the Superclass
- This means that to test the subclass, it must be run not only against the subclass tests, but also against the superclass tests
- Running against the superclass tests is referred to as inheritance testing and ensures that the inheritance has been correctly implemented
- The key problem with inheritance testing is running a set of tests designed for one class against a different class
- Obviously the different class must be compatible (i.e. inherited)

Inheritance Testing – The Problem

- In the standard template we have used this is not possible
- The class to be tested is hard-coded into the test class:

```
1  // Test for class XXX
2  @Test
3  public void test() {
4      XXX x = new XXX();
5      x.doSomething( input0, input1, input2 );
6      assertEquals( x.getResult(), expectedResult );
7  }
```

- On line 4 the test item is created inside the test class: the test can only be run against an object of Class XXX and not a subclass

Poor Solution: Cut-and-Paste

- If we now have a new class to test, that inherits from this class, we need to run these existing tests against the new class (assuming that it is a true subclass that fully supports all the superclass behaviour)

```
1
2  // Inheritance test for class YYY
3  @Test
4  public void test() {
5      XXX x = new YYY();
6      x.doSomething( input0, input1, input2 );
7      assertEquals( x.getResult(), expectedResult );
  }
```

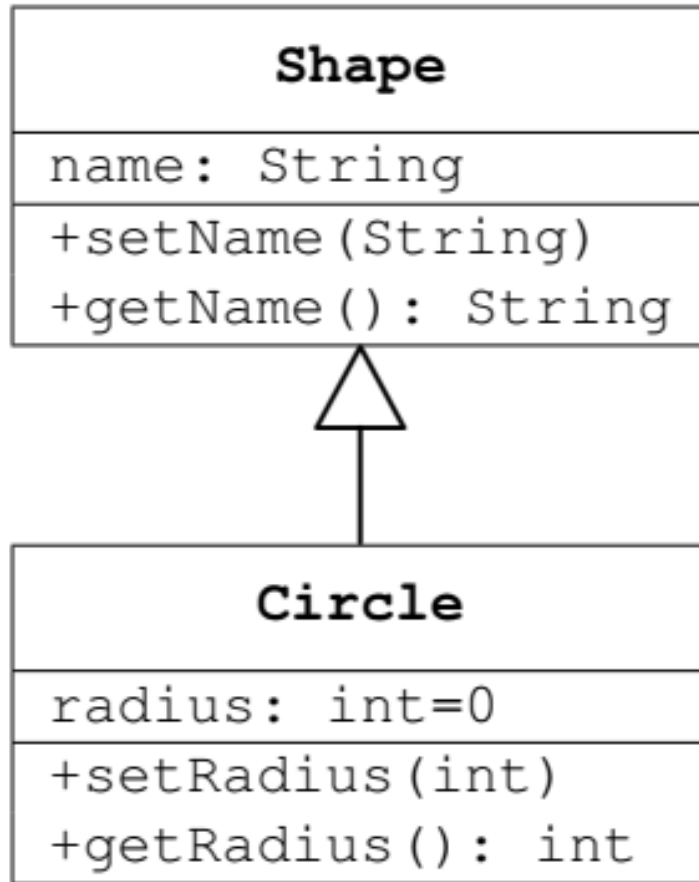
Cut-and-Paste Limitations

- The obvious problem, as with all cut-and-paste approaches, is that any changes to the XXX test code do not automatically get propagated to the YYY test code
 - And it is probably in a different Java file
 - This is a particular problem in modern development methods, where classes are refactored and added to on a regular basis
- There is a second, less obvious problem
 - If a third class ZZZ inherits from YYY, then you need to copy the XXX and YYY tests into the ZZZ test class
 - This results in an explosion of copied code, and is very likely to lead to mistakes and untested code

Cut-and-Paste Strengths

- It is very clear what tests are being run against which class in the class hierarchy
- It is possible to select which tests are to be run for inheritance testing. The tester may wish to not run some test cases either for performance reasons, or because the subclass is not fully Liskov substitutable and the tests are not applicable
- If the YYY constructor takes different parameters from the XXX constructor it is easy to handle – the correct parameters can be just passed to the YYY constructor

Example: Classes Shape and Circle

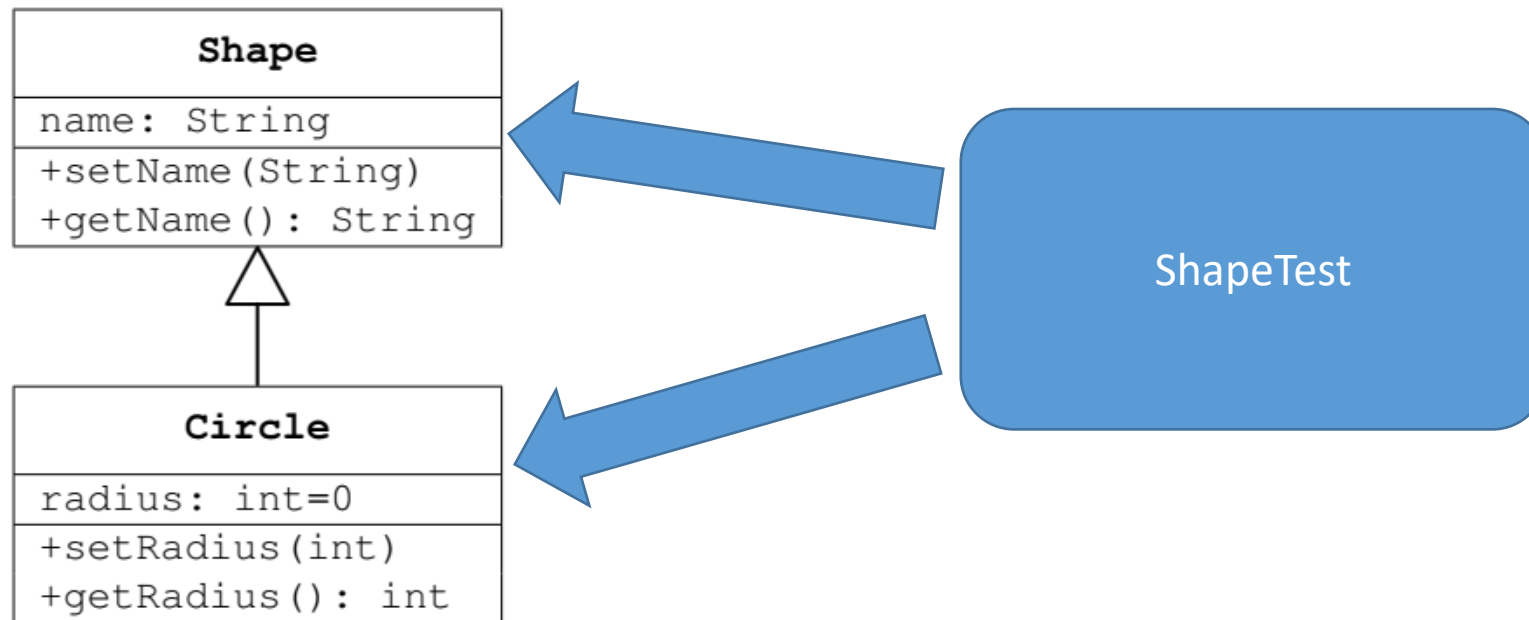


```
class Shape {
    String name="unknown";
    String getName() { return name; }
    void setName(String name) { this.name = name; }
}
```

```
class Circle extends Shape {
    int radius=0;
    int getRadius() { return radius; }
    void setRadius(int radius) { this.radius = radius; }
}
```

Inheritance Testing of Circle

- We want to run the Shape tests against an instance of class Circle



Option 1: Using the Class Name

- In ShapeTest, instead of hard-coding the class to be tested, using the new Shape() constructor
- In this example, the class name is passed as a parameter using **-D** on command line
- A **factory method** creates the object to test

```
public class ShapeTest {  
  
    // Factory method to create a Shape  
    public Shape createShape() throws Exception {  
        String cn=System.getProperty("classname");  
        Class<?> c = Class.forName(cn);  
        Shape o = (Shape) (c.getDeclaredConstructor().newInstance());  
        System.out.println("Running Shape test against instance of  
            "+o.getClass());  
        return o;  
    }  
  
    @Test  
    public void test_demo() throws Exception {  
        Shape o = createShape();  
        o.setName("Test name 1");  
        assertEquals( o.getName(), "Test name 1" );  
    }  
  
}
```

Using the Class Name

- Using a factory method allows objects of **different** classes to be returned by the factory method
- They must be compatible with Shape

```
public class ShapeTest {  
  
    // Factory method to create a Shape  
    public Shape createShape() throws Exception {  
        String cn=System.getProperty("classname");  
        Class<?> c = Class.forName(cn);  
        Shape o = (Shape) c.getDeclaredConstructor().newInstance();  
        System.out.println("Running Shape test against instance of  
            "+o.getClass());  
        return o;  
    }  
  
    @Test  
    public void test_demo() throws Exception {  
        Shape o = createShape();  
        o.setName("Test name 1");  
        assertEquals( o.getName(), "Test name 1" );  
    }  
  
}
```

Using the Class Name

- Using a factory method allows objects of **different** classes to be returned by the factory method
- They must be castable to Shape

```
public class ShapeTest {  
  
    // Factory method to create a Shape  
    public Shape createShape() throws Exception {  
        String cn=System.getProperty("classname");  
        Class<?> c = Class.forName(cn);  
        Shape o = (Shape) c.getDeclaredConstructor().newInstance();  
        System.out.println("Running Shape test against instance of  
            "+o.getClass());  
        return o;  
    }  
  
    @Test  
    public void test_demo() throws Exception {  
        Shape o = createShape();  
        o.setName("Test name 1");  
        assertEquals( o.getName(), "Test name 1" );  
    }  
  
}
```

Using the Class Name

- **Line 13:** gets the class name
- Passed as a Java *property* (using `-D` on the java command line)
- Rather than as a command line parameter
- In TestNG the test class does not have a `main()` method to pass parameters to

```
9 public class ShapeTest {
10
11     // Factory method to create a Shape
12     public Shape createShape() throws Exception {
13         String cn=System.getProperty("classname");
14         Class<?> c = Class.forName(cn);
15         Shape o = (Shape) (c.getDeclaredConstructor().newInstance());
16         System.out.println("Running Shape test against instance of
17                             "+o.getClass());
17         return o;
18     }
}
```

Using the Class Name

- Line 13: get class name
- **Line 14:** finds the class associated with the (full) classname – this loads the .class file

```
9 public class ShapeTest {
10
11     // Factory method to create a Shape
12     public Shape createShape() throws Exception {
13         String cn=System.getProperty("classname");
14         Class<?> c = Class.forName(cn);
15         Shape o = (Shape) (c.getDeclaredConstructor().newInstance());
16         System.out.println("Running Shape test against instance of
                               "+o.getClass());
17         return o;
18     }
```


Using the Class Name

- Line 13: get class name
- Line 14: load the class
- **Line 15:** instantiates an object of that class, by calling the constructor indirectly via the `newInstance()` method – and casts it to a `Shape`

```
9 public class ShapeTest {
10
11     // Factory method to create a Shape
12     public Shape createShape() throws Exception {
13         String cn=System.getProperty("classname");
14         Class<?> c = Class.forName(cn);
15         Shape o = (Shape) (c.getDeclaredConstructor().newInstance());
16         System.out.println("Running Shape test against instance of
17                             "+o.getClass());
17         return o;
18     }
```

Using the Class Name

- Line 13: get class name
- Line 14: load the class
- Line 15: instantiates an object
- **Line 16:** prints out the test name and the classname – providing a record in the test log of what class was tested

```
9 public class ShapeTest {
10
11     // Factory method to create a Shape
12     public Shape createShape() throws Exception {
13         String cn=System.getProperty("classname");
14         Class<?> c = Class.forName(cn);
15         Shape o = (Shape)(c.getDeclaredConstructor().newInstance());
16         System.out.println("Running Shape test against instance of
                               "+o.getClass());
17         return o;
18     }
```

Running ShapeTest on a Shape

```
java -Dclassname=example.Shape  
    -cp testng.jar; jcommander.jar; guice.jar; ./bin  
    org.testng.TestNG  
    -testclass example.ShapeTest  
    -log 2
```

```
Running Shape test against instance of class example.Shape  
PASSED: test_demo  
=====
```

Test Name	Passes	Failures	Skips
test_demo	1	0	0

```
Command line suite  
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0  
=====
```

Running ShapeTest on a Circle

```
java -Dclassname=example.Circle  
-cp testng.jar; jcommander.jar; guice.jar; ./bin  
org.testng.TestNG  
-testclass example.ShapeTest  
-log 2
```

```
Running Shape test against instance of class example.Circle  
PASSED: test_demo  
=====
```

```
Command line suite  
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0  
=====
```

Running CircleTest on a Circle

```
java -Dclassname=example.Circle  
    -cp testng.jar; jcommander.jar; guice.jar; ./bin  
    org.testng.TestNG  
    -testclass example.CircleTest  
    -log 2
```

```
Running CircleTest against instance of class example.Circle  
PASSED: test_circle
```

```
=====
```

Command line test
Tests run: 1, Failures: 0, Skips: 0

```
=====
```

Considerations When Using Classname

- The main disadvantage of this approach for inheritance testing is that it requires the tester to explicitly run the tests for a class against every superclass in the class hierarchy
- The advantage is that the tester can easily select the test classes to execute

Option 2: Inheriting Superclass Tests

- An alternative approach is to make ShapeTest work as an inherited test
- There are many ways to do this – one of the simplest is to use a factory method, which can be overridden by subclass tests

Inheritable Shape Test

- **Lines 12-14** define a factory method that return a Shape
- **Line 13** instantiates a Shape by calling the constructor
- **On line 18**, the factory method is called to create a Shape

```
9  public class ShapeTest {
10
11      // Factory method to create a Shape
12      Shape createInstance() {
13          return new Shape();
14      }
15
16      @Test(groups={"inherited","shape"})
17      public void test_shape() throws Exception {
18          Shape o = createInstance();
19          System.out.println("Running Shape test against instance of
                               "+o.getClass());
20          o.setName("Test name 1");
21          assertEquals( o.getName(), "Test name 1" );
22      }
23
24 }
```


Running Inheritable ShapeTest on a Shape

```
Running Shape test against instance of class example.Shape  
PASSED: test_shape
```

```
=====
```

```
Command line suite
```

```
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
```

```
=====
```

Circle Test

- Class CircleTest has **two** test methods
- It inherits the method `test_shape()`
- It defines the method `test_circle()`

```
9  public class CircleTest extends ShapeTest {
10
11      // Factory method to create a Circle
12      Circle createInstance() {
13          return new Circle();
14      }
15
16      // Shape tests are run automatically
17      // New circle tests go here
18      @Test(groups={"inherited","circle"})
19      public void test_circle() throws Exception {
20          Circle o = createInstance();
21          System.out.println("Running Circle test against instance of
22                          "+o.getClass());
23          assertEquals( o.getRadius(), 44 );
24      }
25
26 }
```

How CircleTest Works

- The (annotated @Test) test method ShapeTest.testShape() is inherited from the ShapeTest class

```
16  @Test(groups={"inherited","shape"})
17  public void test_shape() throws Exception {
18      Shape o = createInstance();
19      System.out.println("Running Shape test against instance of
      "+o.getClass());
20      o.setName("Test name 1");
21      assertEquals( o.getName(), "Test name 1" );
```

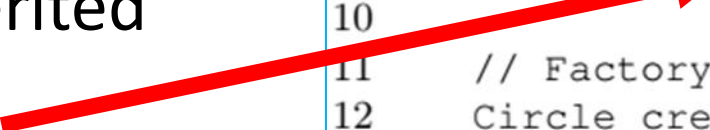
```
9  public class CircleTest extends ShapeTest {
10
11      // Factory method to create a Circle
12      Circle createInstance() {
13          return new Circle();
14      }
15
16      // Shape tests are run automatically
17      // New circle tests go here
18      @Test(groups={"inherited","circle"})
19      public void test_circle() throws Exception {
20          Circle o = createInstance();
21          System.out.println("Running Circle test ag
          "+o.getClass());
22          o.setRadius(44);
23          assertEquals( o.getRadius(), 44 );
24      }
25
26  }
```

How CircleTest Works

- ShapeTest.testShape() is inherited
- When CircleTest is executed, TestNG finds this inherited method, and runs it as a test

```
16  @Test(groups={"inherited","shape"})
17  public void test_shape() throws Exception {
18      Shape o = createInstance();
19      System.out.println("Running Shape test against instance of
        "+o.getClass());
20      o.setName("Test name 1");
21      assertEquals( o.getName(), "Test name 1" );
```

```
9  public class CircleTest extends ShapeTest {
10
11      // Factory method to create a Circle
12      Circle createInstance() {
13          return new Circle();
14      }
15
16      // Shape tests are run automatically
17      // New circle tests go here
18      @Test(groups={"inherited","circle"})
19      public void test_circle() throws Exception {
20          Circle o = createInstance();
21          System.out.println("Running Circle test ag
        "+o.getClass());
22          o.setRadius(44);
23          assertEquals( o.getRadius(), 44 );
24      }
25
26  }
```



How CircleTest Works

- ShapeTest.testShape() is inherited
- ShapeTest.testShape() is run
- The method ShapeTest.testShape() calls createInstance()

```
16  @Test(groups={"inherited","shape"})
17  public void test_shape() throws Exception {
18      Shape o = createInstance();
19      System.out.println("Running Shape test against instance of
      "+o.getClass());
20      o.setName("Test name 1");
21      assertEquals( o.getName(), "Test name 1" );
```

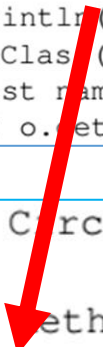
```
9  public class CircleTest extends ShapeTest {
10
11      // Factory method to create a Circle
12      Circle createInstance() {
13          return new Circle();
14      }
15
16      // Shape tests are run automatically
17      // New circle tests go here
18      @Test(groups={"inherited","circle"})
19      public void test_circle() throws Exception {
20          Circle o = createInstance();
21          System.out.println("Running Circle test ag
          "+o.getClass());
22          o.setRadius(44);
23          assertEquals( o.getRadius(), 44 );
24      }
25
26  }
```

How CircleTest Works

- ShapeTest.testShape() is inherited
- ShapeTest.testShape() is run
- ShapeTest.testShape() invokes createInstance()
- As the test is running in CircleTest context, the method CircleTest.createInstance() is called

```
16  @Test(groups={"inherited","shape"})
17  public void test_shape() throws Exception {
18      Shape o = createInstance();
19      System.out.println("Running Shape test against instance of
                             "+o.getClass());
20      o.setName("Test name 1");
21      assertEquals( o.getName(), "Test name 1" );
```

```
9  public class CircleTest extends ShapeTest {
10
11      // Factory method to create a Circle
12      Circle createInstance() {
13          return new Circle();
14      }
15
16      // Shape tests are run automatically
17      // New circle tests go here
18      @Test(groups={"inherited","circle"})
19      public void test_circle() throws Exception {
20          Circle o = createInstance();
21          System.out.println("Running Circle test ag
                             "+o.getClass());
22          o.setRadius(44);
23          assertEquals( o.getRadius(), 44 );
24      }
25
26  }
```



How CircleTest Works

- ShapeTest.testShape() is inherited
- ShapeTest.testShape() is run
- ShapeTest.testShape() invokes createInstance()
- CircleTest.createInstance() is executed
- This returns an object instance of class Circle
- Which is cast to a Shape in test_shape()

```
16  @Test(groups={"inherited","shape"})
17  public void test_shape() throws Exception {
18      Shape o = createInstance();
19      System.out.println("Running Shape test against instance of
        "+o.getClass());
20      o.setName("Test name 1");
21      assertEquals( o.getName(), "Test name 1" );
```

```
9  public class CircleTest extends ShapeTest {
10
11      // Factory method to create a Circle
12      Circle createInstance() {
13          return new Circle();
14      }
15
16      // Shape tests are run automatically
17      // New circle tests go here
18      @Test(groups={"inherited","circle"})
19      public void test_circle() throws Exception {
20          Circle o = createInstance();
21          System.out.println("Running Circle test ag
        "+o.getClass());
22          o.setRadius(44);
23          assertEquals( o.getRadius(), 44 );
24      }
25
26 }
```

How CircleTest Works

- ShapeTest.testShape() is inherited
- ShapeTest.testShape() is run
- ShapeTest.testShape() invokes createInstance()
- CircleTest.createInstance() is executed
- CircleTest.createInstance() returns a Circle
- **This causes shapeTest() to be run against a Circle**

```
16  @Test(groups={"inherited","shape"})
17  public void test_shape() throws Exception {
18      Shape o = createInstance();
19      System.out.println("Running Shape test against instance of
        "+o.getClass());
20      o.setName("Test name 1");
21      assertEquals( o.getName(), "Test name 1" );
```

```
9  public class CircleTest extends ShapeTest {
10
11      // Factory method to create a Circle
12      Circle createInstance() {
13          return new Circle();
14      }
15
16      // Shape tests are run automatically
17      // New circle tests go here
18      @Test(groups={"inherited","circle"})
19      public void test_circle() throws Exception {
20          Circle o = createInstance();
21          System.out.println("Running Circle test ag
        "+o.getClass());
22          o.setRadius(44);
23          assertEquals( o.getRadius(), 44 );
24      }
25
26 }
```


How ShapeTest Works

- When `shapeTest()` runs in `ShapeTest` context, `ShapeTest.createInstance()` is called, which returns a object instance of class `Shape`
- The result of this is that both the shape tests and the circle tests can be run against a `Circle`
- This technique works automatically in a deep inheritance hierarchy, as each subclass test inherits all the superclass tests in the hierarchy

```
Running Circle test against instance of class example.Circle
Running Shape test against instance of class example.Circle
PASSED: test_circle
PASSED: test_shape
=====
Command line suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
=====
```

Considerations When Using Test Inheritance

- Two Issues:
 1. The order of testing – it may be desired to do inheritance testing first
 2. Test selection – perhaps the tester does not want to run all the ShapeTest test methods against a Circle
- TestNG, and other frameworks, provide a number of mechanisms to support these issues
- Examples follow...

Inheritance Test Ordering

- There are a number of ways to enforce ordering
- Specifying **dependencies** is probably the simplest
- Test method dependencies can be used to enforce ordering, causing the inherited Shape tests to be executed prior to the new Circle tests
- This works well with further tests in the test hierarchy – all the inherited test methods will be run from the top of the hierarchy downwards, in the required order

ShapeTest with Dependencies

```
// Shape tests are run automatically
// New circle tests go here
@Test(dependsOnMethods={"test_shape"})
public void test_circle() throws Exception {
    Circle o = createInstance();
    System.out.println("Running Circle test against instance of
        "+o.getClass());
    o.setRadius(44);
    assertEquals( o.getRadius(), 44 );
}
```

```
Running Shape test against instance of class example.Circle
Running Circle test against instance of class example.Circle
PASSED: test_shape
PASSED: test_circle
=====
Command line suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
=====
```

Inheritance Test Selection

- As discussed previously, there may be reasons to exclude some of the inheritance tests
- Three common reasons for this are:
 1. where a tester wants to exclude all the inheritance tests
 2. where the performance is too slow to run all the tests every time in a deep hierarchy
 3. where some subclasses in the hierarchy are not fully Liskov substitutable.
- Several ways to select a subset of tests
- Using **test groups** is probably the simplest approach

Tests With Groups

```
9 public class ShapeTest {
10
11     // Factory method to create a Shape
12     Shape createInstance() {
13         return new Shape();
14     }
15
16     @Test(groups={"inherited","shape"})
17     public void test_shape() throws Exception {
18         Shape o = createInstance();
19         System.out.println("Running Shape test again
20                             "+o.getClass());
21         o.setName("Test name 1");
22         assertEquals( o.getName(), "Test name 1" );
23     }
24 }
```

```
9 public class CircleTest extends ShapeTest {
10
11     // Factory method to create a Circle
12     Circle createInstance() {
13         return new Circle();
14     }
15
16     // Shape tests are run automatically
17     // New circle tests go here
18     @Test(groups={"inherited","circle"})
19     public void test_circle() throws Exception {
20         Circle o = createInstance();
21         System.out.println("Running Circle test again
22                             "+o.getClass());
23         o.setRadius(44);
24         assertEquals( o.getRadius(), 44 );
25     }
26 }
```

Test Output With Groups

- Test groups to be run can be selected at runtime (TestNG parameter)
- If the group **shape** is specified, the tests in group **shape** are run
- If group **circle** is specified, the tests in group **circle** are run

```
Running Circle test against instance of class example.Circle
PASSED: test_circle

=====
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
=====
```

- If group **inherited** is specified, all the tests run in no particular order

```
Running Circle test against instance of class example.Circle
Running Shape test against instance of class example.Circle
PASSED: test_circle
PASSED: test_shape

=====
Command line suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
=====
```

9. Testing Private Methods

- The test class cannot call private methods in the class being tested
- We've already reviewed this in detail
- Some solutions:
 - Temporarily comment out the "private" keyword
 - Use Java Reflection

10. Interfacing to Web Applications

- Selenium is a popular tool for automating web based applications
- We have used it as a representative example of a library enabling automated web based application testing
- Some key features:
 - Starting WebDriver and loading a url
 - Verifying the current page title
 - Finding an element on a page
 - Simulating user input
 - Getting output from the web application
 - Wait for an element to appear

Starting WebDriver and loading a url

- Loading a page whose URL is specified in the String variable *url*

```
// Create web driver (this code uses chrome)
driver = new ChromeDriver();
// Create wait
wait = new WebDriverWait( driver, Duration.ofSeconds(5) );
// Open web page
driver.get( url );
```

Selenium note: ChromeDriver() downloads an executable image depending on the Operating System, and version of the Chrome in use.

There are separate 'webdrivers' for Chrome, Chromium, Firefox, Edge, etc.

Verify, Find, and Typing

- Verifying the current page title

```
assertEquals("<expected name>", driver.getTitle() );
```

- Finding an element on a page by id

```
driver.findElement(By.id("<elementid>"))
```

- Simulate data being typed into an input field

```
driver.findElement(By.id("<elementid>")).  
    sendKeys("<input data>");
```

Output, Clicking, and Checkboxes

- Get a value from an input field (disabled input fields are often used for output)

```
driver.findElement(By.id("<elementid>")).  
    getAttribute("value")
```

- Simulate a user clicking on an element in a page

```
driver.findElement(By.id("<elementid>")).click()
```

- Verify if a checkbox is selected

```
assertTrue(driver.findElement(By.id("<elementid>")).  
    isSelected())
```

Waiting For An Element or Page To Appear

- Important for web applications, as there can be a significant delay (in the order of several seconds) before the page updates
- Reference the Selenium documentation for all the ExpectedConditions:
 - “visibilityOfElement”: web element is actually visible on screen
 - “presenceOfElement”: web element is present in the DOM

```
wait.until(ExpectedConditions.visibilityOfElementLocated(  
    By.id("<expected element>")));
```

- Wait for a page to be displayed

```
wait.until(ExpectedConditions.titleIs("<expected title>"));
```

11. Interfacing to Desktop Applications

- Java GUI desktop applications are usually developed using Java AWT, Swing, JavaFX, or other libraries
- As for web applications, the key issues are:
 - how to find the GUI elements to interact with
 - how to interact with them
- Easy to write a Selenium-style test library (based on AWT)
 - By, find, click etc.
- Or other cross-platform test tools, such as QF-TEST, etc.
- Or platform-specific tools (e.g. WinAppDriver)

Typical Desktop App Structure

- Two main approaches:
 - "extends JFrame"
 - "new JFrame()"

Typical Desktop App Test Structure

- Startup:
 - Call the **main()** method in the application to start it running
 - or call **new Classname()** if no main

Typical Desktop App Test Structure

- Finding the application window:
 - Call **Window.getWindows()** to find the application window

Typical Desktop App Test Structure

- Finding the application window:
 - Call **Window.getWindows()** to find the application window
 - Call **javax.swing.FocusManager.getCurrentManager().getActiveWindow()** to find the active window after starting the application

Typical Desktop App Test Structure

- GUI event thread
 - In AWT, all GUI events are handled on a separate thread
 - Place all calls that interact directly with the GUI on the "GUI event thread"
 - Use **Swing.InvokeAndWait()** for a Swing application

Typical Desktop App Test Structure

- Finding a window (screen) name:
 - Call **JFrame.getTitle()** to get the title of a window

Typical Desktop App Test Structure

- Finding the components of a window (or container):
 - Call **Container.getComponents()** recursively to find contained components (window, button, textbox, etc)

Typical Desktop App Test Structure

- Finding a component name:
 - Use **Component.getName()** to get component name (not id)

Typical Desktop App Test Structure

- Interact with the components:
 - **JButton.doClick()**, **JTextBox.setText()**, **JTextBox.getText()**, etc
 - Or use **Java AWT Robot** for lower level interactions (x&y location, click, keypress)
 - On the GUI event thread
 - Check the documentation
 - 'Reads' may not need to be on this thread
 - 'Writes' usually do

Key Differences from Web Applications

- A desktop application has full control of the application screen, whereas for a web page the user can return to a stale window by typing in the URL or using the browser back button
- Reference GUI library APIs for details
- There are a number of frameworks for Java GUI application testing
 - built on top of the underlying AWT/Swing/JavaFX libraries
 - Many OS-dependent GUI test frameworks
 - Many language-independent GUI test frameworks

12. Interfacing to Mobile Applications

- Developing tests for mobile applications is similar to developing tests for web applications
- Some example tools are:
 - selendroid (Selenium for Android) – API as for Selenium, using the selendroid-client libraries
 - IOSDriver on Apple phones – based on Selenium/WebDriver
- Many other tools
- View an up-to-date comparisons of these online

This Afternoon's Lab

- Test Automation
- Instructions & ZIP file
- Quiz