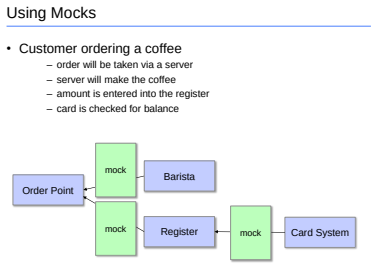
Java Unit Testing Tools

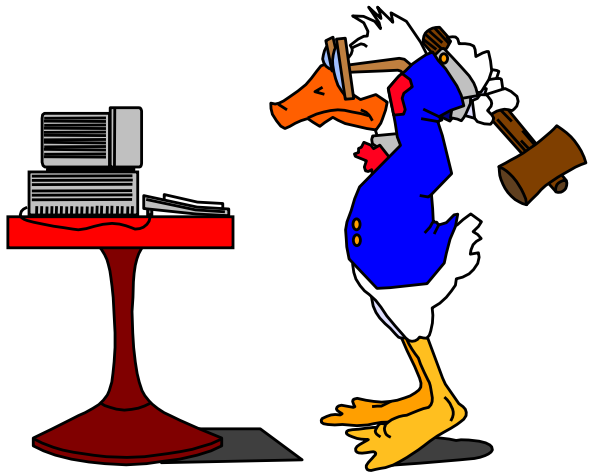
Java Programming
November 2020



Unit Testing

- Testing and debugging small units of work
  - class
  - method

- Find and fix errors in logic early
  - less cost to fix
  - unit testing catches 65% of defects found in testing

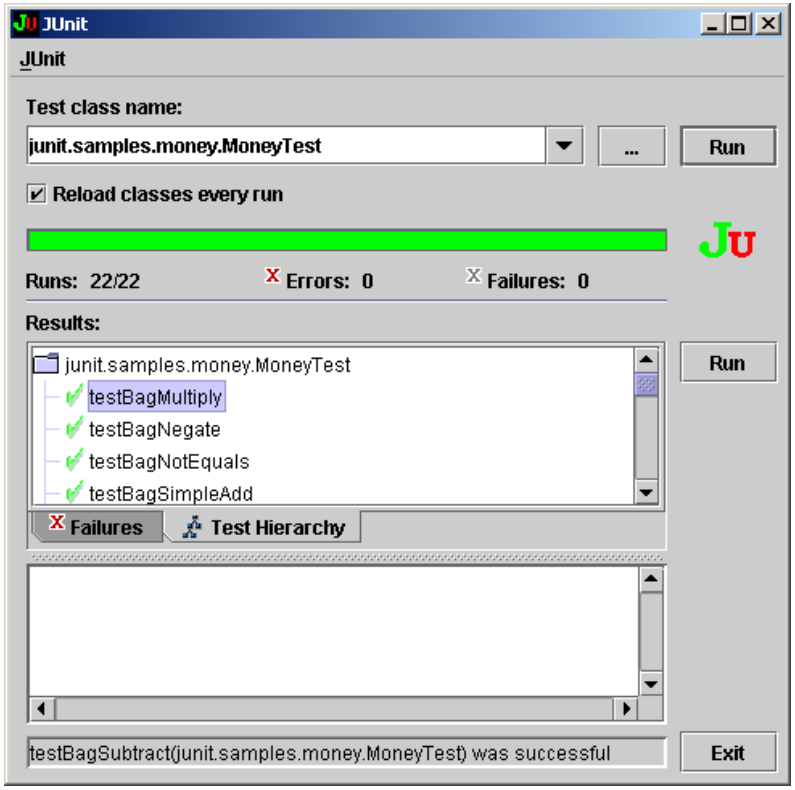


Running Unit Tests

- Test Drivers
  - control overall test
  - provide inputs
- Stubs
  - take the place of other modules referenced from the module being tested
- Automation
  - ability to execute tests without direct intervention
  - shell scripts or batch files
  - specific testing tools

JUnit

- Java based framework for writing unit tests
- Written by Kent Beck
  - defined Extreme Programming (XP) methodology
- and Erich Gamma
  - "Design Patterns”
- Integrates with other tools
  - Gradle
  - Maven
  - IntelliJ
  - Eclipse



JUnit

- Java framework for Unit Testing
  - available for other languages NUnit, CppUnit etc.
  - www.junit.org

- Annotation based specification of tests
  - @Test, @SuiteClasses, @Before, @After, @Ignore
  - assertions provided by static methods
  - can be extended by libraries such as Hamcrest

- Terminology
  - Assertions used to test results
  - Text Class – a class with some unit tests in it
  - Test Fixture – supporting operation for 1 or more tests
  - Test (or Test Method) – a test implemented in a test Class
  - Test Suite – a set of tests grouped together
  - Test Harness / Runner – The tool that actually executes the tests

Using JUnit 4

- Annotate a class with @RunWith(JUnit4.class)
  - can specify other runners, e.g. SpringJUnit4ClassRunner.class
  - suites are run using @RunWith(Suite.class)
- Annotate any public void methods with @Test
  - no need to name method test_...()
- Add @Before, @After to methods for setup/tear down
  - will be run before and after each @Test
  - can also use @BeforeClass, @AfterClass on static methods

Example JUnit 4 Test Case

```
package junit.samples;
import org.junit.*;

// Do some simple tests.

@RunWith(JUnit4.class)

public class SimpleTest {

  protected int fValue1;
  protected int fValue2;

  @Before

  public void setUp() {
    fValue1= 2;
    fValue2= 3;
  }

  public static void main (String[] args) {
     org.junit.runner.JUnitCore.runClasses(
          SimpleTest.class);
  }

  // test methods to follow…
```

No base class, @RunWith annotation defines how to launch

@Before annotation defines method to be before each @Test method

Example JUnit 4 Test Case

```
  @Test
  public void addTwoValue() {
    double result= fValue1 + fValue2;
    org.junit.Assert.assertTrue(result == 5);
  }

  @Test(expected=ArithmeticException.class)
  public void divideByZero() {
    int zero= 0;
    int result= 8/zero;
  }

  @Ignore
  @Test
  public void notYetWritten() {
    // need to write this test
  }
}
```

use @Test to mark test methods

use @Test(expected) to declare what exception a method must throw to pass the test

JUnit 4 Test Suites

- @Suite annotation used to define classes in suite

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestLoginFeature.class,
    TestLogoutFeature.class
})
public class FeatureSuite() {
   // no need for implementation

}
```

Hamcrest Assertions Example

```
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.*;

...

assertThat("good", is(equalTo("good")));
assertThat("good", is(not("bad")));
assertThat(new Object(), is(not(sameInstance(new Object()))));

// also supports collection matchers
List<String> list = Arrays.asList("c#", "c++", "javascript");
assertThat(list, hasItems("c#", "javascript"));
assertThat(list, everyItem(containsString("c")));

// can combine multiple assertions
assertThat("javascript", both(containsString("c"),
                              containsString("java")));

// can support messages
Integer anInt = Integer.valueOf(42);
assertThat("should be the same", anInt, anInt);
```

Testing and Exceptions

- JUnit test cases are based around exceptions
  - failed assertion causes exception to be thrown
  - other exception will also cause test to fail

- How to test a method that throws an exception?
  - or constructor?

```
public class Account {
 private int balance;
 public Account(int initBalance) {
  if ( initBalance < 0 ) {
   throw new IllegalArgumentException(
                "Initial balance cannot be negative");
  }
  this.balance = initBalance;
 }
 public int getBalance() { return this.balance; }
}
```

Testing and Exceptions

- Testing this functionality:
  - if valid balance is passed, then the Account object is created correctly

```
…
   @Test
   public void testCreation() {
     Account a = new Account(100);
     assertEquals(a.getBalance(), 100);
   }
…
```

Testing and Exceptions

- Testing this functionality:
  - if valid balance is passed, then the Account object is created correctly
  - if negative balance is passed, exception should be thrown

```
…
   @Test
   public void testCreation() {
     Account a = new Account(100);
     assertEquals(a.getBalance(), 100);
   }

   @Test( expected = IllegalArgumentException.class )
   public void testCreationWithNegativeBalance() {
     Account a = new Account(-100);
   }
…
```

Testing and Checked Exceptions

- Checked exceptions require different handling
  - test case must declare that it throws exception

```
public class OverdrawnException extends Exception {

  public OverdrawnException( String msg ) { super(msg); }

}
```

```
…
  @Test ( expected = OverdrawnException.class)
  public void testWithdrawSignalsOverdrawn()
                          throws OverdrawnException {
    Account a = new Account(100);
    a.withdraw(200);
  }
…
```

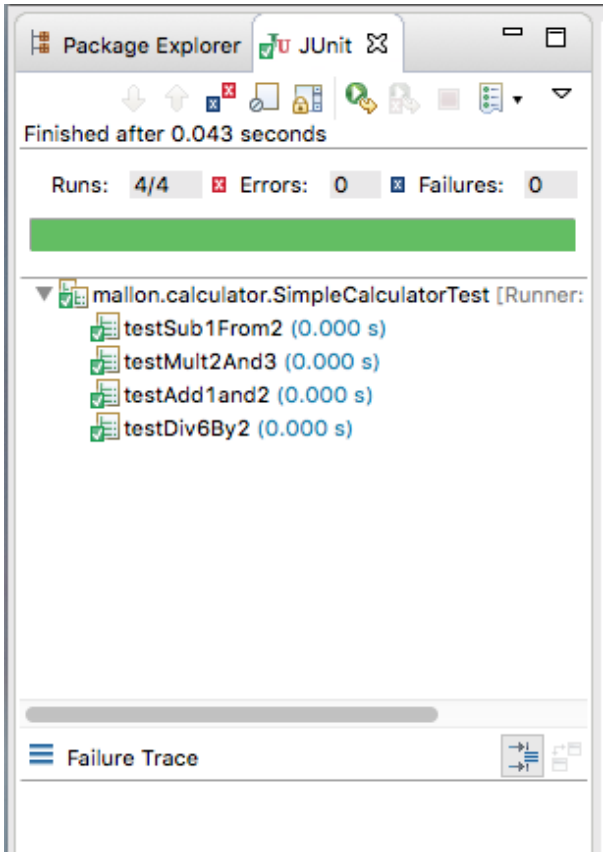
Testing and Checked Exceptions

- Alternative approach is to handle the exception
  - allows properties of the exception to be checked
  - also possible for unchecked exceptions

```
…
   @Test
   public void testWithdrawTooMuchThrowsException() {
     Account a = new Account(100);
     try {
       a.withdraw(200);
       fail("OverdrawnException should have been thrown");
     } catch ( OverdrawnException e ) {
       assertTrue(e.getMessage().contains("oops"));
     }
   }
…
```

Using JUnit in Eclipse

- JUnit directly supported by many IDEs



- JUnit specified in project build files (e.g. Gradle or Ivy)
  - ensures correct version
  - allows for tests to run in CI environment

Test Doubles and Mock Objects

- There are several kinds of test doubles
  - dummy objects
  - fake objects
  - mock objects
  - stubs
- Mock objects verify behavior
  - were the right calls made in the right order?
  - stubs only tend to verify state
- TDD often uses mock objects
  - define feature to be implemented
  - write tests
  - use mocks for external interactions

Mock Objects

- Mock objects have the same interface as the real thing
  - they check the context of each call
  - they can contain assertions
- Mock object provide
  - canned responses to method calls
  - typically based on parameter values
- Mock frameworks help you work with mock objects
  - NMock, Rhino, Moq, NSubstitute for C#
  - Mockito, EasyMock for Java
  - MockPP, googlemock for C++
  - Test::MockObject for Perl
  - ngMock for JavaScript / AngularJS
  - unittest.mock for Python

Mock Objects

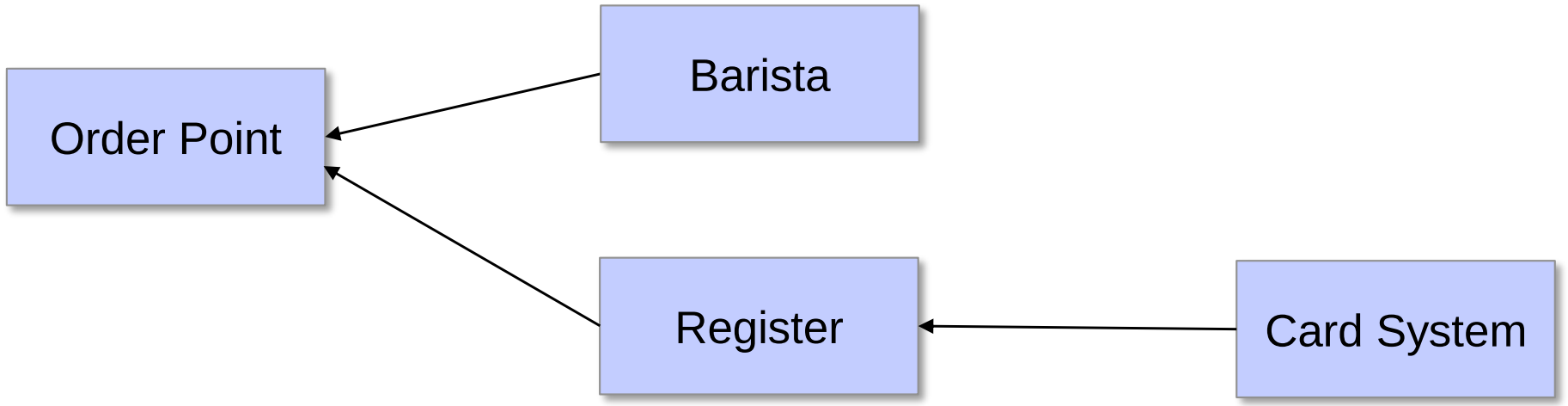
- Used when impossible or impractical to use the real thing
  - e.g. testing a servlet outside the container
- Typical situations leading to a mock object:

- Object supplies non-deterministic results
  - e.g. the current time or the current temperature
- Object has states that are difficult to create or reproduce
  - e.g. a network error
- Real thing is slow
  - e.g. a complete database,
  - requiring initialization before tests
- It does not yet exist or may change behavior

- Would have to include test specific information /methods

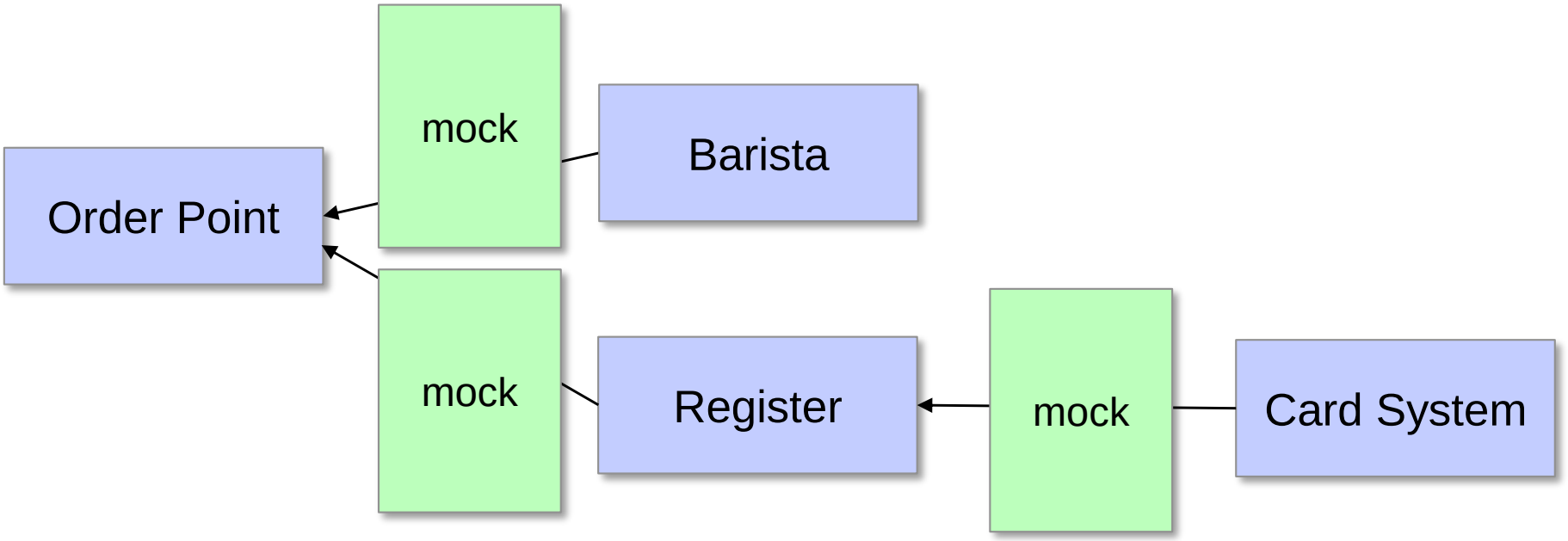
Using Mocks

- Customer ordering a coffee
  - order will be taken via a server
  - server will make the coffee
  - amount is entered into the register
  - card is checked for balance



Using Mocks

- Customer ordering a coffee
  - order will be taken via a server
  - server will make the coffee
  - amount is entered into the register
  - card is checked for balance



Using Mockito

- Set up for the test
  - create the object to test and the mock object
  - hook up a reference to a mock

```
import org.junit.Test;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

public class CalculatorTest {

    @Test
    public void shouldReturnNewTotal3() {
        DataService mock = mock(DataService.class);
        when(mock.getData()).thenReturn(2);

        Calculator calc = new Calculator(mock);

        int result = calc.add(1);
        assertEquals(3, result);
    }
}
```

DataService mock created and behavior defined

Using Mockito

- Tests typically involve
  - telling the mock object what to expect and using the object
  - verifying usage was correct

```
import org.junit.Test;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

public class DocumentServiceTest {
    @Test
    public void test() {
        DocumentService mock = mock(DocumentService.class);
          when(mock.addDocument(
                        "jzh", "Document1")).thenReturn(true);

          Documentor doc = new Documentor("jzh", mock);
          boolean result = doc.addDocument("Document1");
          assertTrue("Documentor expected to return true", result);

          verify(mock).addDocument("jzh", "Document1");
    }
}
```

Mockito Mock annotation

- Use the `@Mock` annotation
  - also needs the `MockitoJUnitRunner` to initialize the mocks
  - can pass in via the constructor or via a setter

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;
import static org.hamcrest.MatcherAssert.assertThat;

@RunWith(MockitoJUnitRunner.class)
public class TestOrderPoint {

   @Mock
   private Server server;

   @Mock
   private Register register;

   private OrderPoint orderPoint;

   @Before
   public void before() {
      orderPoint = new OrderPoint(register, server);
   }
```

Mockito Mock annotation

```
@Test
public void the_server_should_make_the_requested_beverage() {
   Beverage beverage = new Beverage("coffee");
   Card card = new Card();
   orderPoint.order(beverage, card);
   verify(server).make(beverage);
}

@Test
public void when_the_payment_is_confirmed_the_order_is_accepted() {
   Beverage beverage = new Beverage("coffee");
   Card card = new Card();
   when(register.checkPayment(
      Mockito.any(Card.class))).thenReturn(true);

   boolean orderStatus = orderPoint.order(beverage, card);
   assertThat(orderStatus, is(equalTo(true)));
}
```

Over Mocking

- It is possible to over mock
  - can result in testing the mocks
  - with very little production code tested
- Avoid more than 2 or 3 mocks per test
  - otherwise gets hard to manage
  - and probably implies some design issues
  - e.g. class under test may have too many responsibilities
  - or insufficient encapsulation
- Only mock your nearest neighbor
  - try to avoid mocking dependencies of dependencies
  - can result in increasingly complex mocks