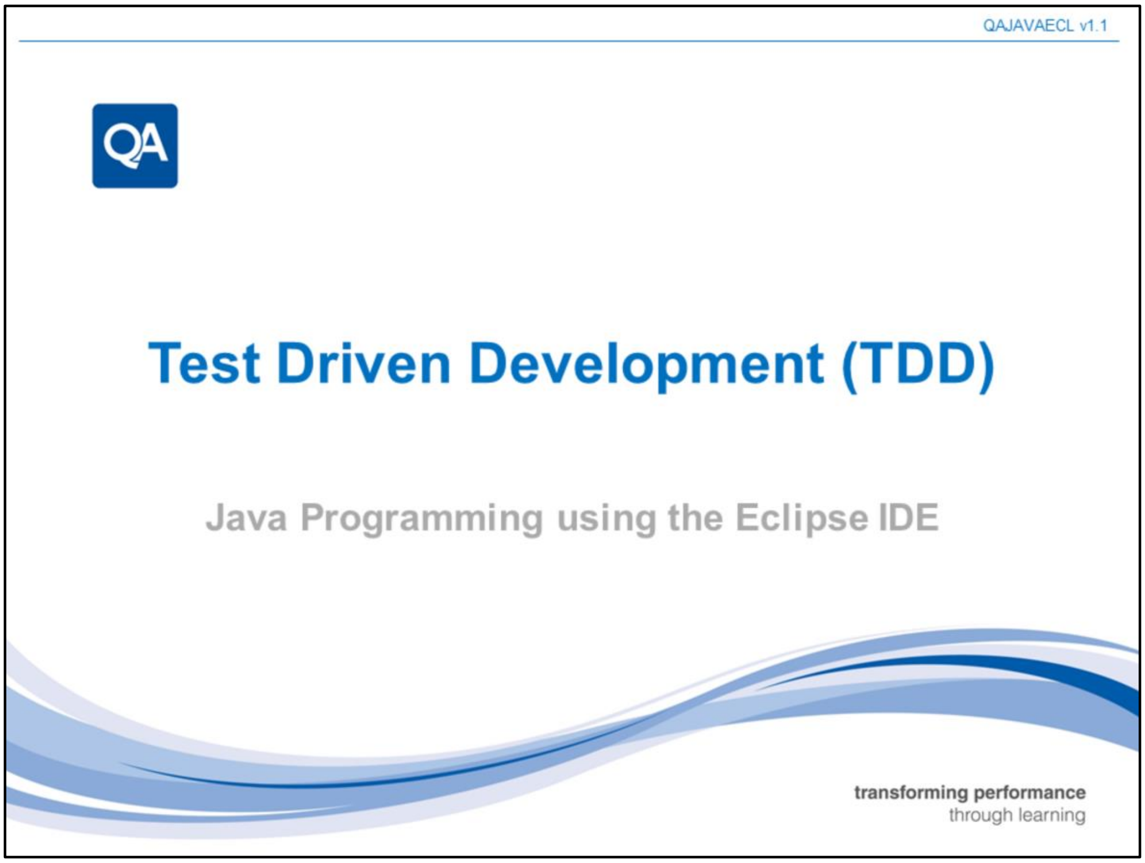
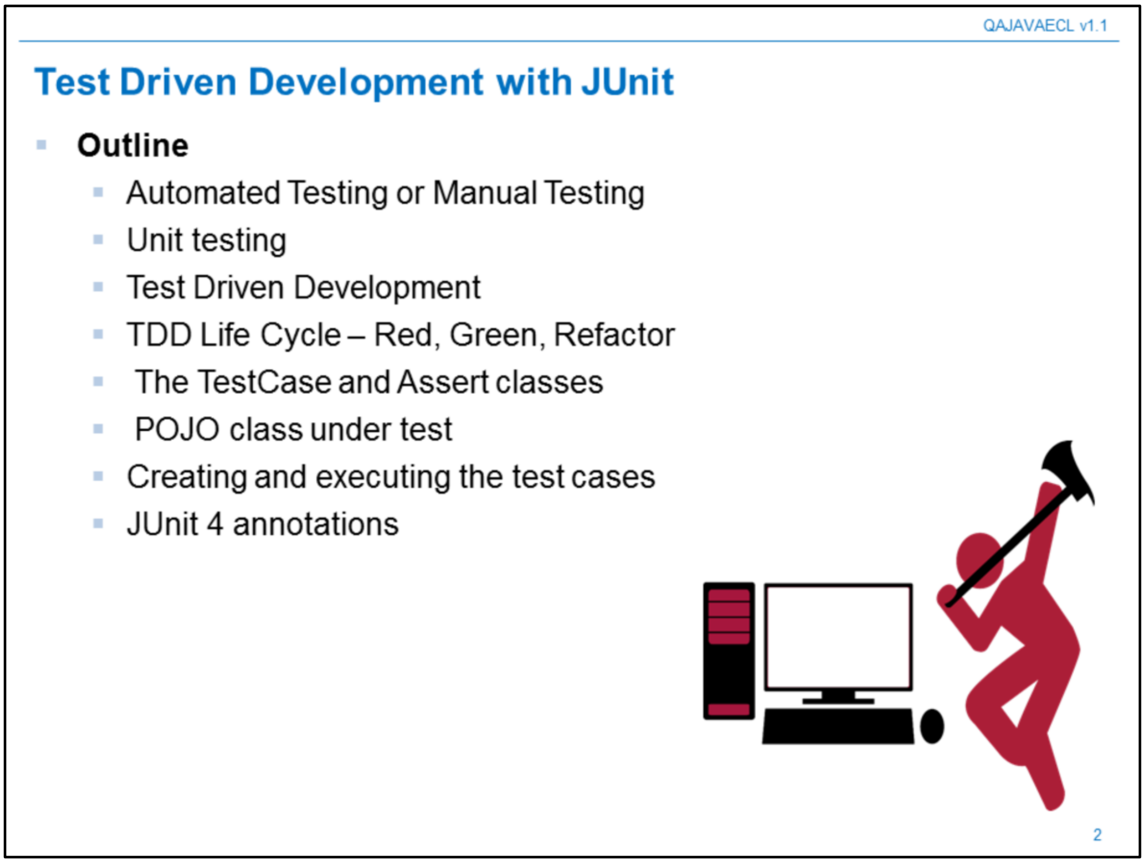


QAJAVAECL v1.1

Test Driven Development (TDD)

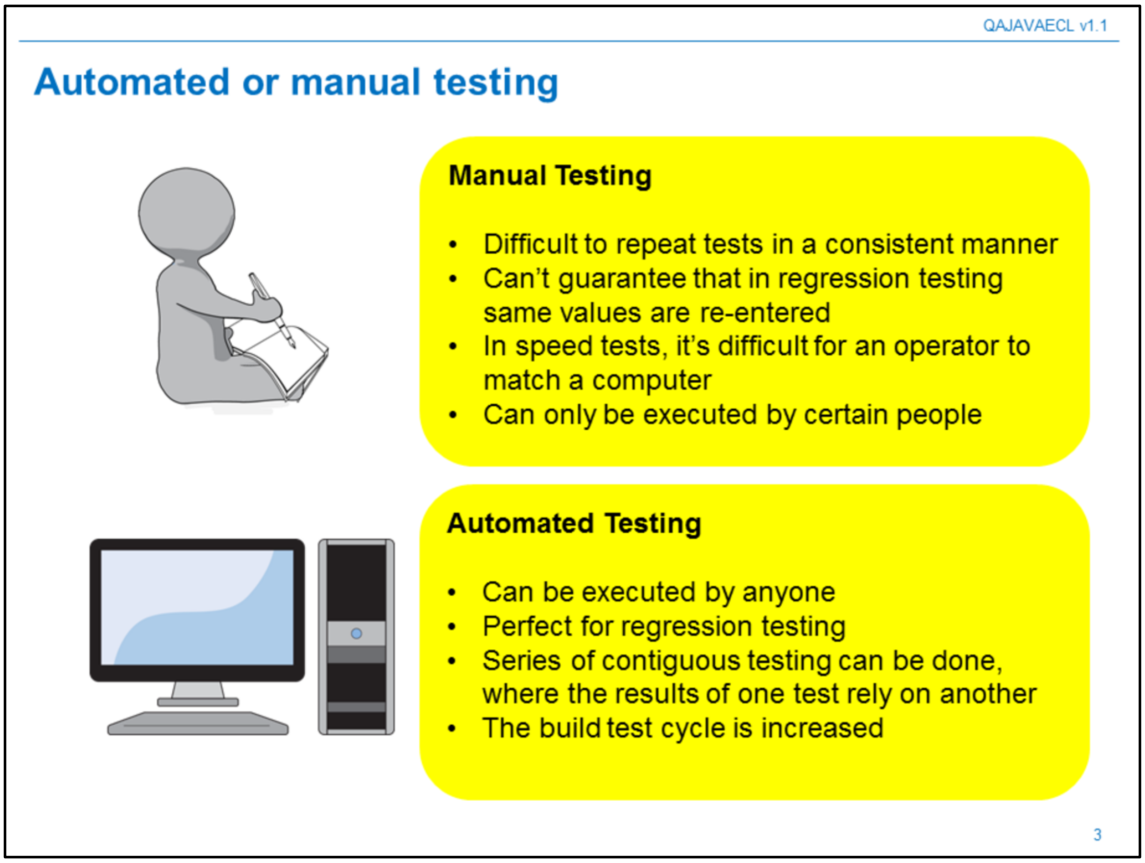
Java Programming using the Eclipse IDE

transforming performance
through learning



Test Driven Development with JUnit

- **Outline**
  - Automated Testing or Manual Testing
  - Unit testing
  - Test Driven Development
  - TDD Life Cycle – Red, Green, Refactor
  - The TestCase and Assert classes
  - POJO class under test
  - Creating and executing the test cases
  - JUnit 4 annotations



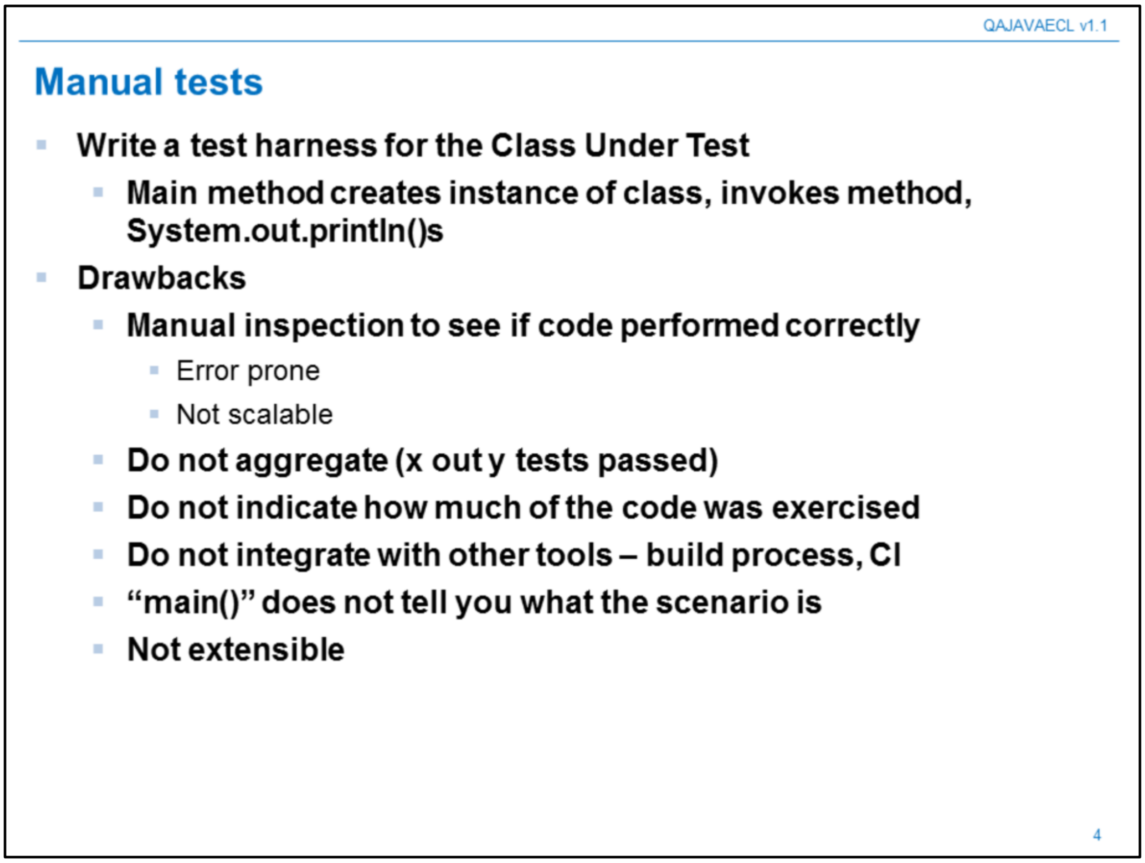
Automated or manual testing

Manual Testing

- Difficult to repeat tests in a consistent manner
- Can't guarantee that in regression testing same values are re-entered
- In speed tests, it's difficult for an operator to match a computer
- Can only be executed by certain people

Automated Testing

- Can be executed by anyone
- Perfect for regression testing
- Series of contiguous testing can be done, where the results of one test rely on another
- The build test cycle is increased



QAJAVAECL v1.1

Manual tests

- Write a test harness for the Class Under Test
  - Main method creates instance of class, invokes method, System.out.println()s
- Drawbacks
  - Manual inspection to see if code performed correctly
    - Error prone
    - Not scalable
  - Do not aggregate (x out y tests passed)
  - Do not indicate how much of the code was exercised
  - Do not integrate with other tools – build process, CI
  - “main()” does not tell you what the scenario is
  - Not extensible

4

Visual inspection of console output is inherently risky (requires the person knows what is supposed to be printed out, and reads it correctly). This is not going to scale to 100s of tests.

Anyone should be able to run the tests at the click of a button and know whether they have all passed or not.

In short:

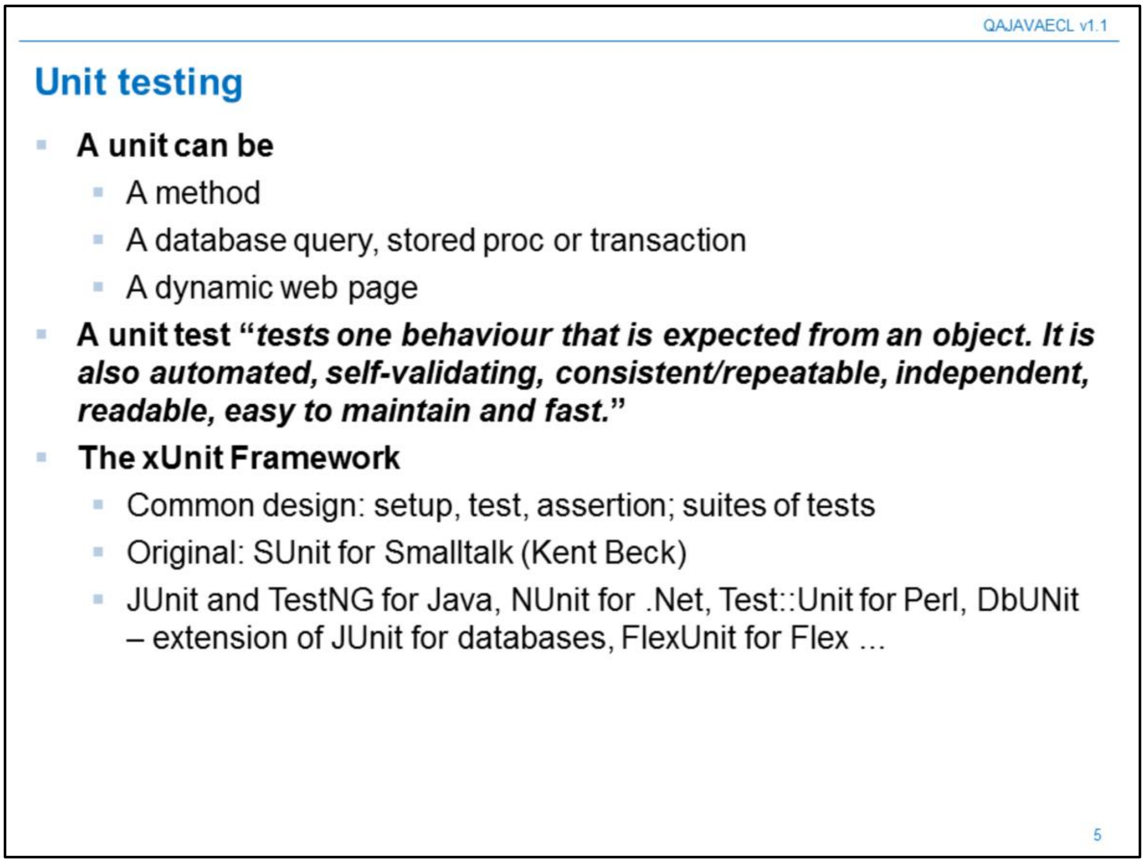
There is no framework (Roy Osherove, The Art of Unit Testing, p. 23)

> There’s no framework we can hook up with other tools, such as coverage, to give measures of how much of the application code was exercised
>
> There’s no framework we might want to extend if we’re testing code involving databases, or outputting html or xml

Acronyms sometimes used:

SUT: System Under Test

CUT: Class Under Test



QAJAVAECL v1.1

Unit testing

- **A unit can be**
  - A method
  - A database query, stored proc or transaction
  - A dynamic web page
- **A unit test "*tests one behaviour that is expected from an object. It is also automated, self-validating, consistent/repeatable, independent, readable, easy to maintain and fast.*"**
- **The xUnit Framework**
  - Common design: setup, test, assertion; suites of tests
  - Original: SUnit for Smalltalk (Kent Beck)
  - JUnit and TestNG for Java, NUnit for .Net, Test::Unit for Perl, DbUNit – extension of JUnit for databases, FlexUnit for Flex …

5

Unit Testing is also known as Component or Module Testing. The concept originated in the 1970s, and the initial concept of what constitutes a "unit" was probably quite open-ended. It is a form of "White Box" testing – in other words, where the test has complete knowledge of the internals of the component being tested. This contrasts with "Black Box" testing, in which only the interface of the component is being tested.

Robert Martin (Clean Code, Ch. 9) captures the modern conception of unit tests in the acronym FIRST:

Fast: they need to be fast so that developers will run them very frequently

Independent: one test must not be dependent on, or be affected by, any other test

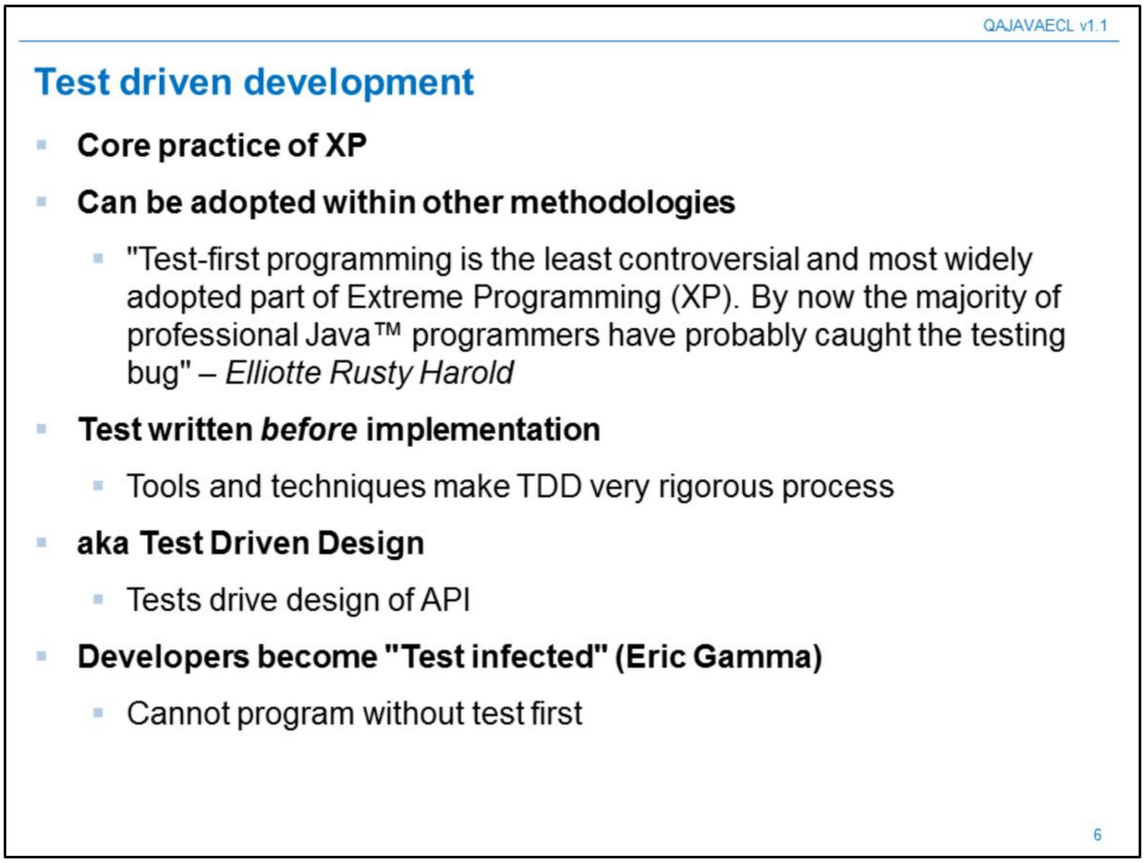
Repeatable: they must be repeatable in every environment: development, QA, production

Self-Validating: they should either pass or fail automatically, not require any manual intervention or manual inspection of output logs

Timely: they should be written in a timely fashion (just before the application code – i.e. TDD)

For a list of xUnit implementations, see

http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks


QAJAVAECL v1.1

Test driven development

- **Core practice of XP**
- **Can be adopted within other methodologies**
  - "Test-first programming is the least controversial and most widely adopted part of Extreme Programming (XP). By now the majority of professional Java™ programmers have probably caught the testing bug" – *Elliotte Rusty Harold*
- **Test written *before* implementation**
  - Tools and techniques make TDD very rigorous process
- **aka Test Driven Design**
  - Tests drive design of API
- **Developers become "Test infected" (Eric Gamma)**
  - Cannot program without test first

6

For me, it is inconceivable that in today's world any serious developer would develop an application without using a modern IDE with powerful refactoring capabilities, and develop their code in a largely test-driven manner. The risks of not driving development through tests are too great.

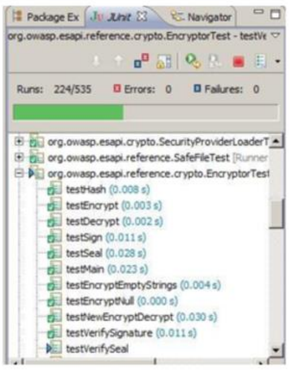
When you've done TDD, you will be familiar with this: "I'll just make this little change, I know it's not going to have any consequences"… and then you re-run the tests, only to find that some tests are now red. "Oh yes, of course… that change led to this consequence, which led to that now being wrong."

If you hadn't got the suite of tests covering your code base, that little bug would have insinuated itself into your application, only to come to light some months later, requiring someone to spend a lot of time tracking it down, with who knows how many other repercussions for other bits of the code.

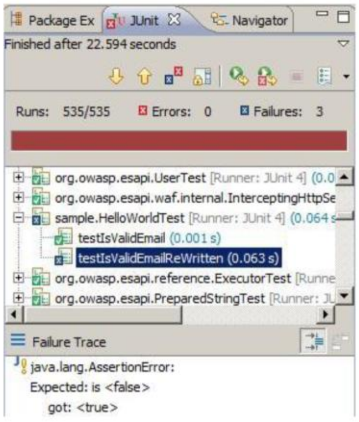
So, when you become 'test infected', when you have some new bit of functionality to implement, you don't just dive in and start coding straight away. You sit back and ask yourself: "So, how am I going to specify my next step in a test?" And sometimes you may find yourself not being 'productive' in a conventional sense, thinking about your tests, discussing them with your colleagues, reorganising your tests, breaking large test classes up, extracting common setup to superclasses etc. And then suddenly you may be able to write many new tests, and then surprisingly quickly implement the code which they specify.

Test driven development

- **Test – Code – Refactor**
- **Kent Beck's summary of TDD:**
  - 1. Write new code only if you first have a failing automated test
  - 2. Eliminate duplication
- **Red – Green – Refactor**



```
org.owasp.esapi.reference.crypto.EncryptorTest - testVe
Runs:   224/535     Errors:   0     Failures:   0
org.owasp.esapi.crypto.SecurityProviderLoaderT
org.owasp.esapi.reference.SafeFileTest [Runner
org.owasp.esapi.reference.crypto.EncryptorTest
    testHash (0.008 s)
    testEncrypt (0.003 s)
    testDecrypt (0.002 s)
    testSign (0.011 s)
    testSeal (0.028 s)
    testMain (0.023 s)
    testEncryptEmptyStrings (0.004 s)
    testEncryptNull (0.000 s)
    testNewEncryptDecrypt (0.030 s)
    testVerifySignature (0.011 s)
    testVerifySeal
```

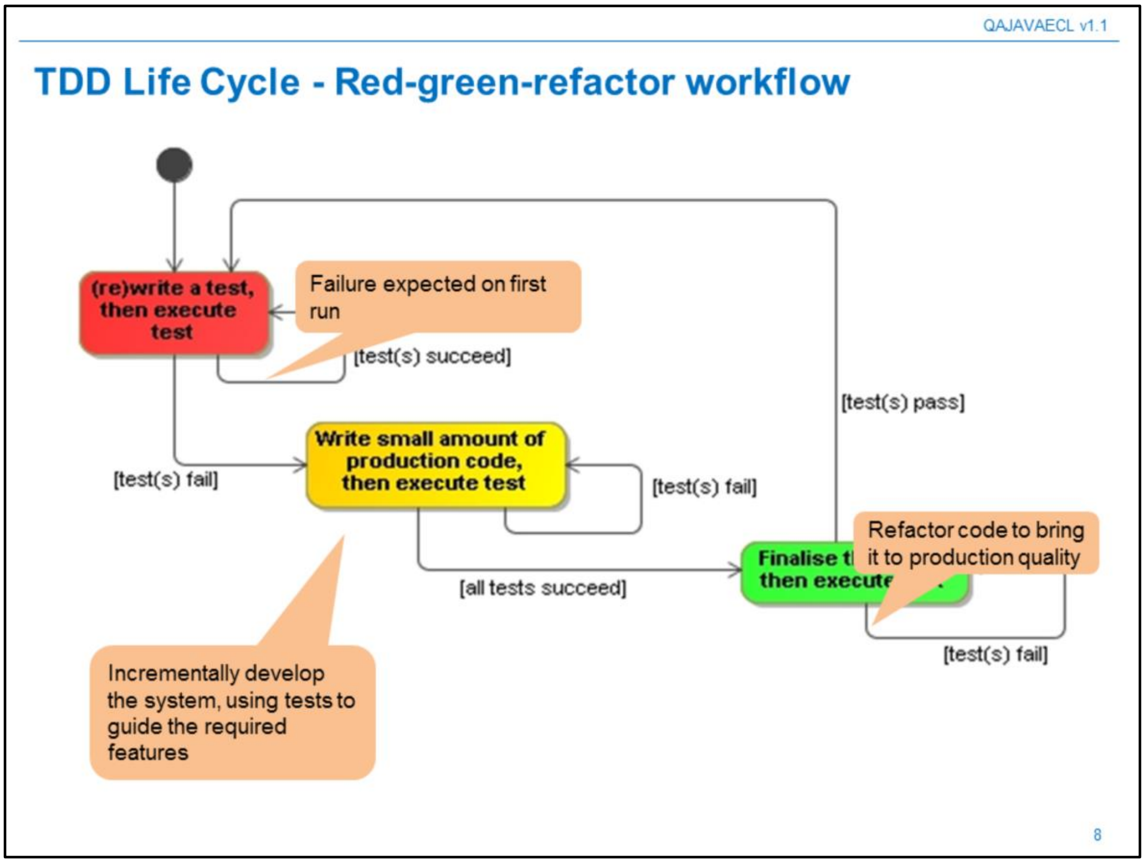


```
Finished after 22.594 seconds
Runs:   535/535     Errors:   0     Failures:   3
org.owasp.esapi.UserTest [Runner: JUnit 4] (0.0
org.owasp.esapi.waf.internal.InterceptingHttpSe
sample.HelloWorldTest [Runner: JUnit 4] (0.064 s
    testIsValidEmail (0.001 s)
    testIsValidEmailReWritten (0.063 s)
org.owasp.esapi.reference.ExecutorTest [Runne
org.owasp.esapi.PreparedStringTest [Runner: JU

java.lang.AssertionError:
  Expected: is <false>
     got: <true>
```

7

The graphic on the left illustrates that the famous green bar is indeed a progress bar. The image on the right shows the final outcome of running this series of tests: there are 3 failures.

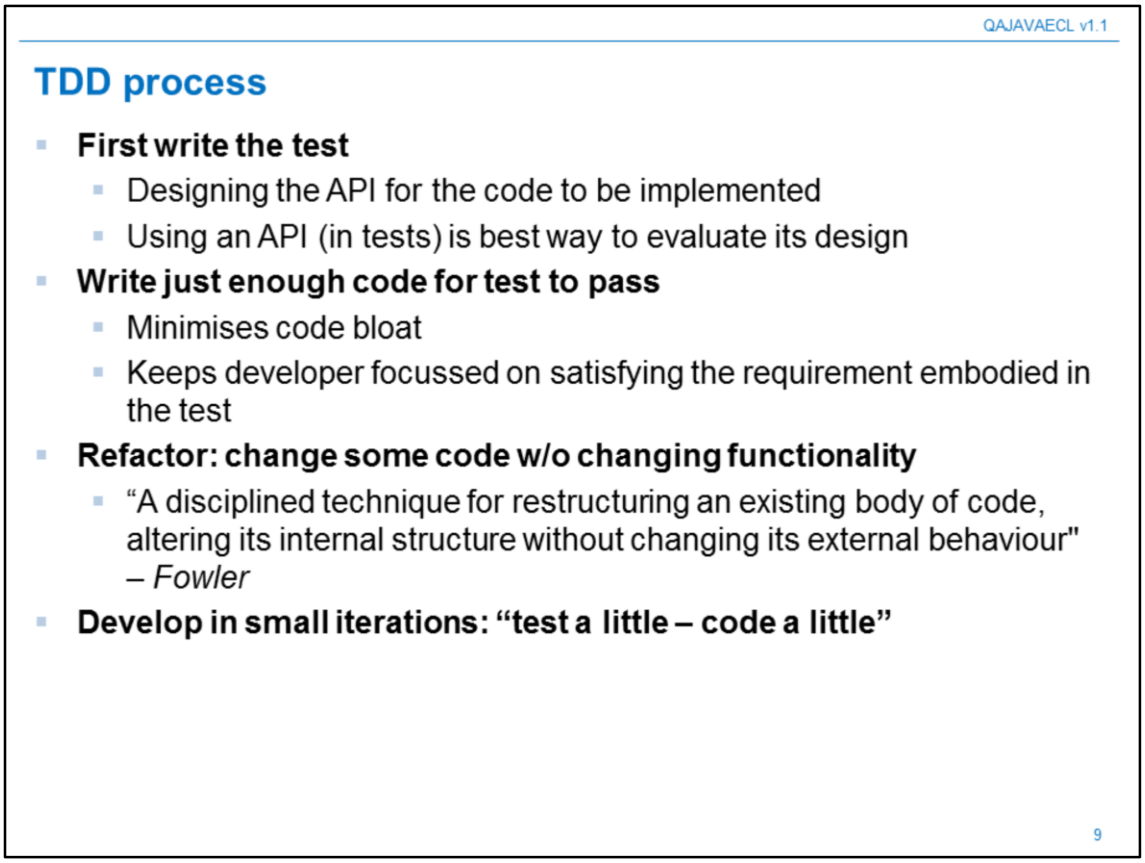


QAJAVAECL v1.1

TDD Life Cycle - Red-green-refactor workflow

(re)write a test, then execute test
Failure expected on first run
[test(s) succeed]
[test(s) pass]
Write small amount of production code, then execute test
[test(s) fail]
[test(s) fail]
Finalise t then execute
Refactor code to bring it to production quality
[all tests succeed]
[test(s) fail]
Incrementally develop the system, using tests to guide the required features

8

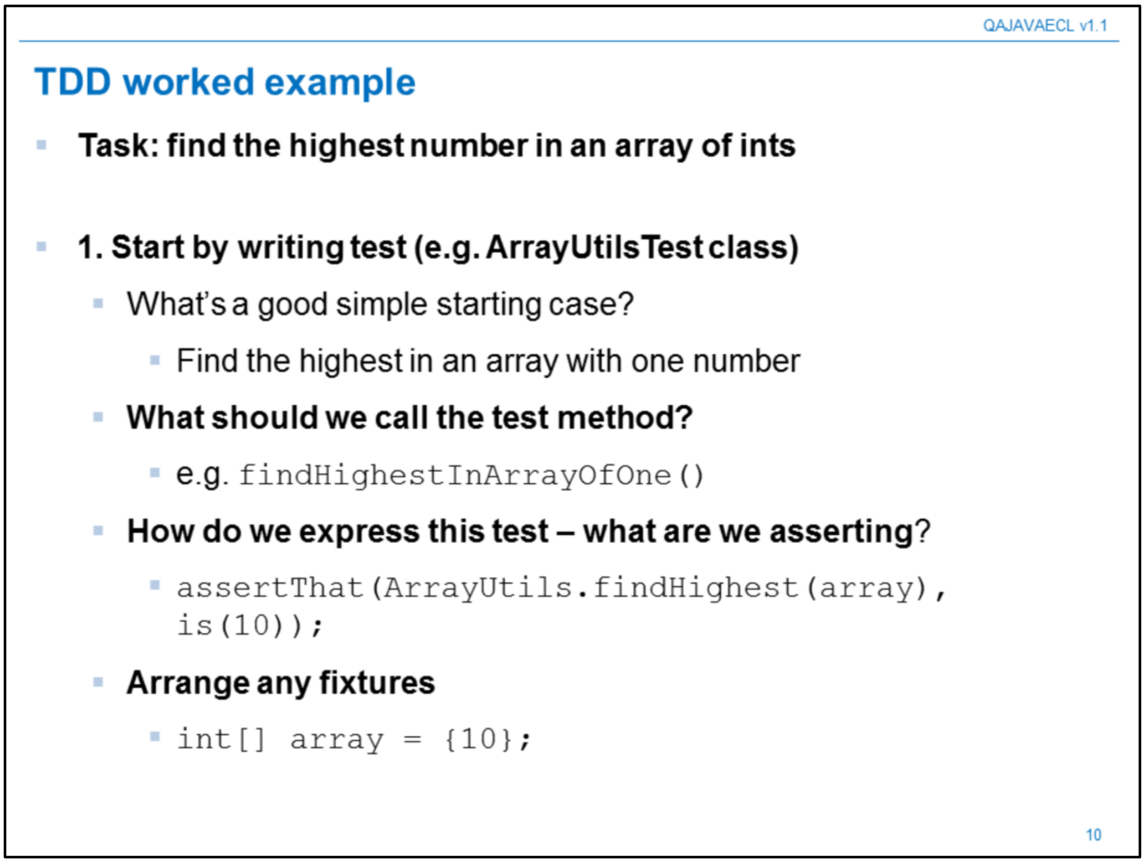


QAJAVAECL v1.1

TDD process

- **First write the test**
  - Designing the API for the code to be implemented
  - Using an API (in tests) is best way to evaluate its design
- **Write just enough code for test to pass**
  - Minimises code bloat
  - Keeps developer focussed on satisfying the requirement embodied in the test
- **Refactor: change some code w/o changing functionality**
  - “A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour" – *Fowler*
- **Develop in small iterations: “test a little – code a little”**

9



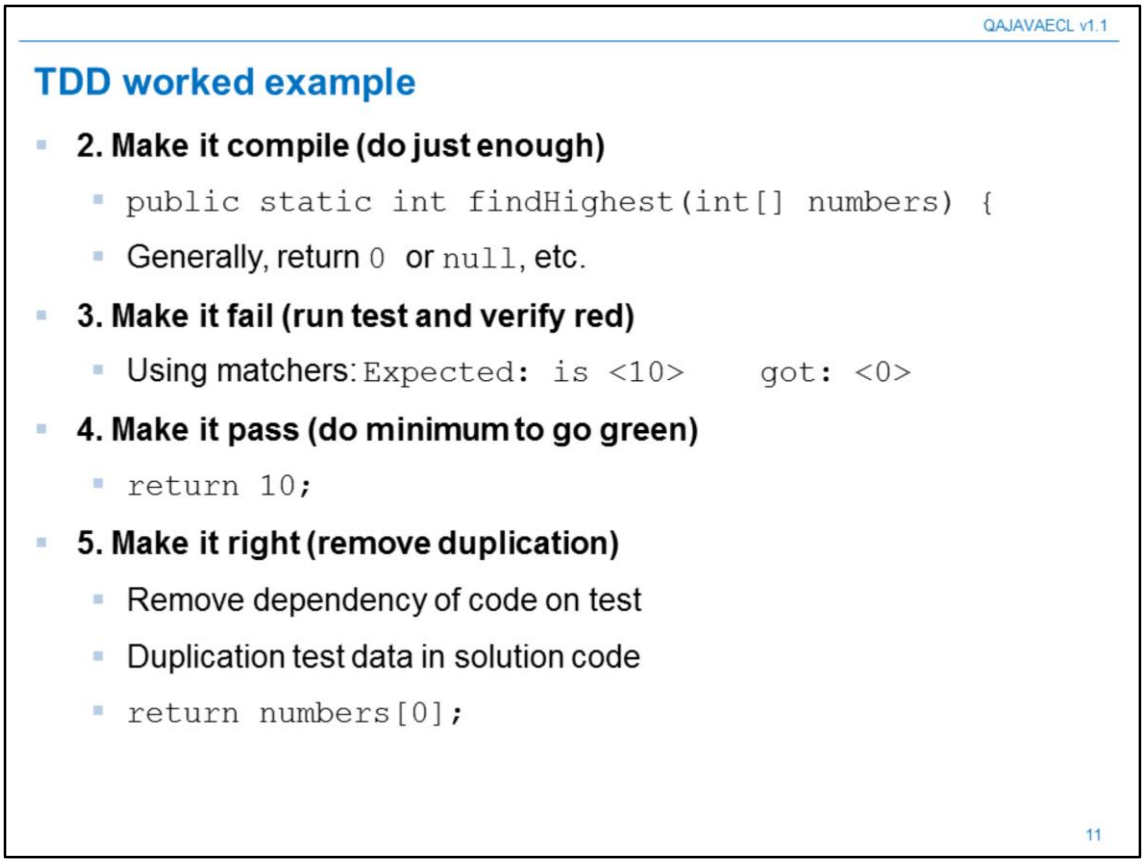
QAJAVAECL v1.1

TDD worked example

- **Task: find the highest number in an array of ints**

- **1. Start by writing test (e.g. ArrayUtilsTest class)**
  - What's a good simple starting case?
    - Find the highest in an array with one number
  - **What should we call the test method?**
    - e.g. `findHighestInArrayOfOne()`
  - **How do we express this test – what are we asserting**?
    - `assertThat(ArrayUtils.findHighest(array), is(10));`
  - **Arrange any fixtures**
    - `int[] array = {10};`

10



QAJAVAECL v1.1

TDD worked example

- **2. Make it compile (do just enough)**
  - `public static int findHighest(int[] numbers) {`
  - Generally, return `0` or `null`, etc.
- **3. Make it fail (run test and verify red)**
  - Using matchers: `Expected: is <10>     got: <0>`
- **4. Make it pass (do minimum to go green)**
  - `return 10;`
- **5. Make it right (remove duplication)**
  - Remove dependency of code on test
  - Duplication test data in solution code
  - `return numbers[0];`

11

At step 3 we run the test and get JUnit's Red bar. In TDD, this constitutes success: we have completed the first essential step, writing a failing test.

At step 4 we do the least to get the test to pass – run the test again: Green bar.

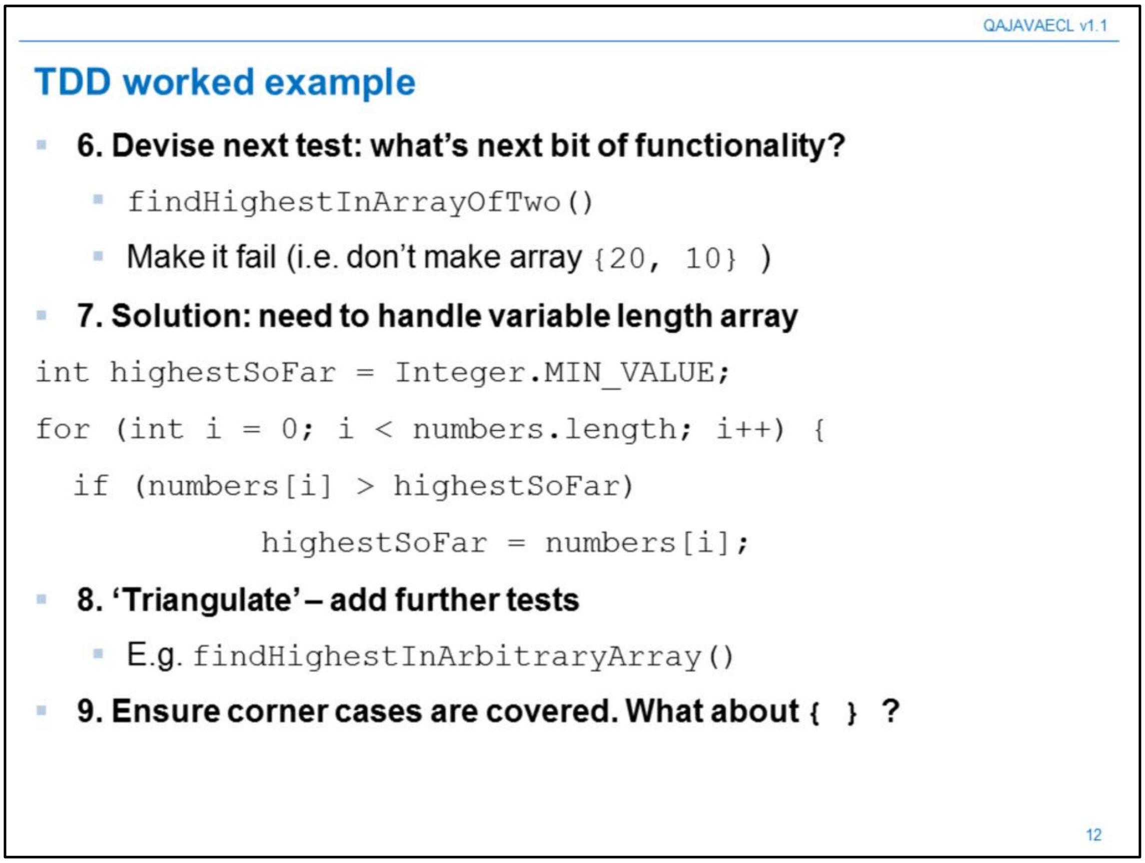
Kent Beck writes, "Steve Freeman pointed out that the problem with the test and code as it sits is not duplication ... The problem is the dependency between the code and the test—you can’t change one without changing the other. Our goal is to be able to write another test that “makes sense” to us, without having to change the code, something that is not possible with the current implementation.

... If dependency is the problem, duplication is the symptom."

-from Test-Driven Development By Example

At step 5 ask yourself:

- Have we specified some functionality in the form of a test?
- Have we satisfied what the test requires?


QAJAVAECL v1.1

TDD worked example

- **6. Devise next test: what's next bit of functionality?**
  - `findHighestInArrayOfTwo()`
  - Make it fail (i.e. don't make array `{20, 10}`)
- **7. Solution: need to handle variable length array**

```
int highestSoFar = Integer.MIN_VALUE;
for (int i = 0; i < numbers.length; i++) {
  if (numbers[i] > highestSoFar)
            highestSoFar = numbers[i];
```

- **8. 'Triangulate' – add further tests**
  - E.g. `findHighestInArbitraryArray()`
- **9. Ensure corner cases are covered. What about `{ }` ?**

12

What should we do about the empty array?

Return an int? Any value we could provide, e.g. 0, or Integer.MIN_VALUE, would be incorrect

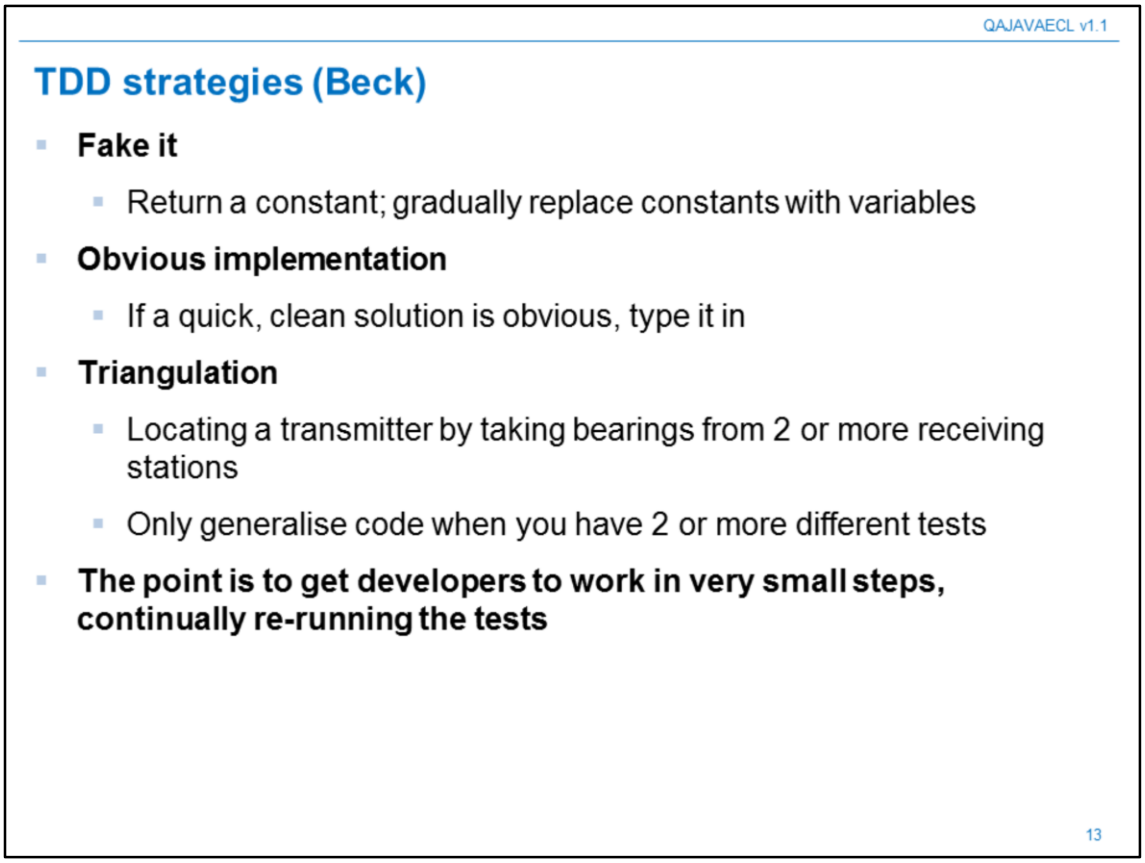
Throw a checked exception? It seems a bit extreme to make ordinary clients handle this exception if this is a rare and unusual case

Throw a runtime exception. Start by specifying with the annotations syntax:

```
@Test(expected=RuntimeException.class)
public void findHighestInEmptyArrayThrows() {
    int [] array = {};
    ArrayUtils.findHighest(array);
}
```

Then if appropriate, refine to the more explicit format:

```
@Test public void findHighestInEmptyArrayThrowsWithMsg() {
    int [] array = {};
    try {
        ArrayUtils.findHighest(array);
        fail("An exception should have been thrown");
    } catch (RuntimeException e) {
        assertThat(e.getMessage(), is("Empty array"));
    }
```



QAJAVAECL v1.1

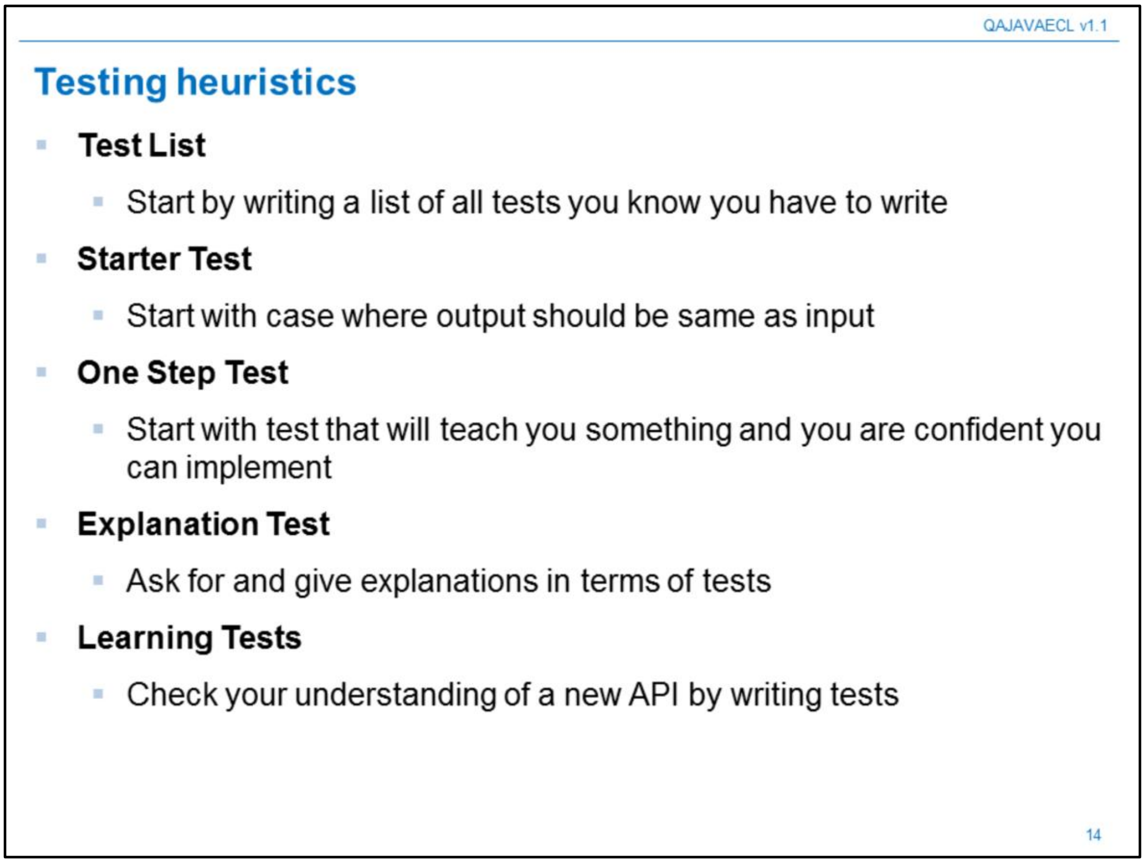
TDD strategies (Beck)

- **Fake it**
  - Return a constant; gradually replace constants with variables
- **Obvious implementation**
  - If a quick, clean solution is obvious, type it in
- **Triangulation**
  - Locating a transmitter by taking bearings from 2 or more receiving stations
  - Only generalise code when you have 2 or more different tests
- **The point is to get developers to work in very small steps, continually re-running the tests**

13

```
        <=> A
        /  \
       /    \
      /      \
     /        \
    /          \
   /            \
  ¶              []
  B               C
```

Triangulation: someone on the boat at A can determine their position on a chart by taking compass bearings to the lighthouse at B and the tower at C. Or conversely, observers at B and C can work out the location of the boat by taking bearings to it from their known points on the shore and sharing their readings. In fact, sailors are taught to take a three-point fix, with charted landmarks that are as widely separated as possible, for better accuracy.



QAJAVAECL v1.1

Testing heuristics

- **Test List**
  - Start by writing a list of all tests you know you have to write
- **Starter Test**
  - Start with case where output should be same as input
- **One Step Test**
  - Start with test that will teach you something and you are confident you can implement
- **Explanation Test**
  - Ask for and give explanations in terms of tests
- **Learning Tests**
  - Check your understanding of a new API by writing tests

14

These mainly come from Kent Beck's Test Driven Development, Chapter 26 – "Red Bar Patterns". They're primarily suggestions for breaking down a seemingly mountainous task of developing some new functionality in a test-driven way, into very small, tractable steps.

For example:

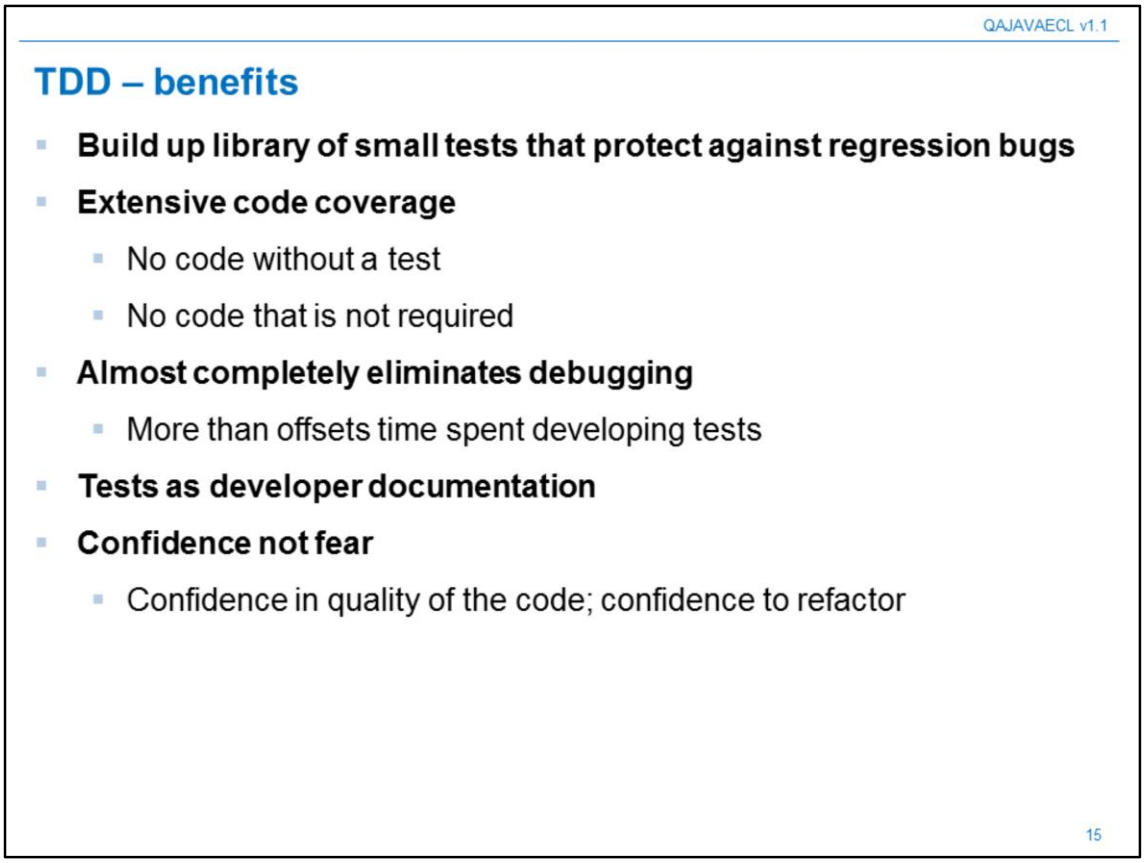
"a poster on the Extreme Programming newsgroup asked about how to write a polygon reducer test-first. The input is a mesh of polygons and the output is a mesh of polygons that describes precisely the same surface, but with the fewest possible polygons. “How can I test-drive this problem since getting a test to work requires reading Ph.D. theses?”

Starter Test provides an answer:

The output should be the same as the input. Some configurations of polygons are already normalised, incapable of further reduction.

The input should be as small as possible, like a single polygon, or even an empty list of polygons."

Explanation Test is primarily for communicating within the development group, clarifying the requirements for some item of functionality by expressing them precisely in terms of tests.



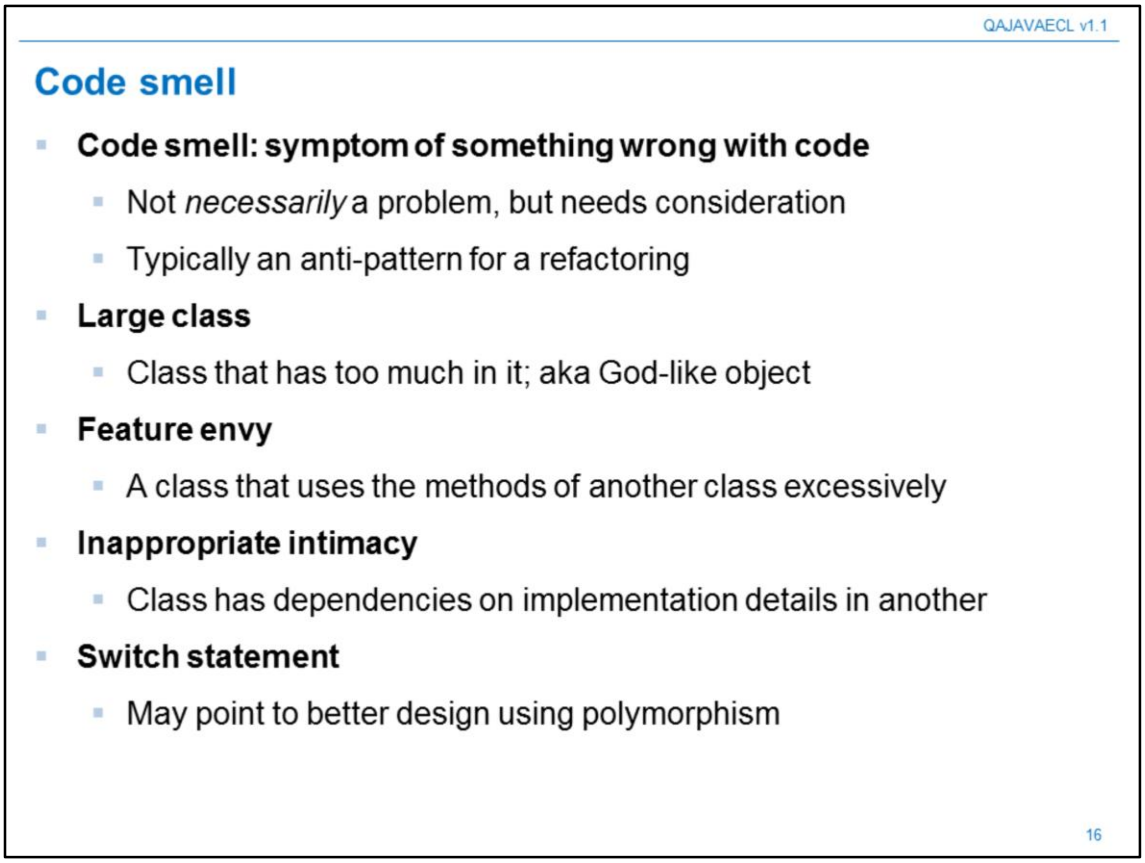
QAJAVAECL v1.1

TDD – benefits

- **Build up library of small tests that protect against regression bugs**
- **Extensive code coverage**
  - No code without a test
  - No code that is not required
- **Almost completely eliminates debugging**
  - More than offsets time spent developing tests
- **Tests as developer documentation**
- **Confidence not fear**
  - Confidence in quality of the code; confidence to refactor

15

A regression bug is a defect which stops some bit of functionality working, after an event such as a code release, or refactoring.

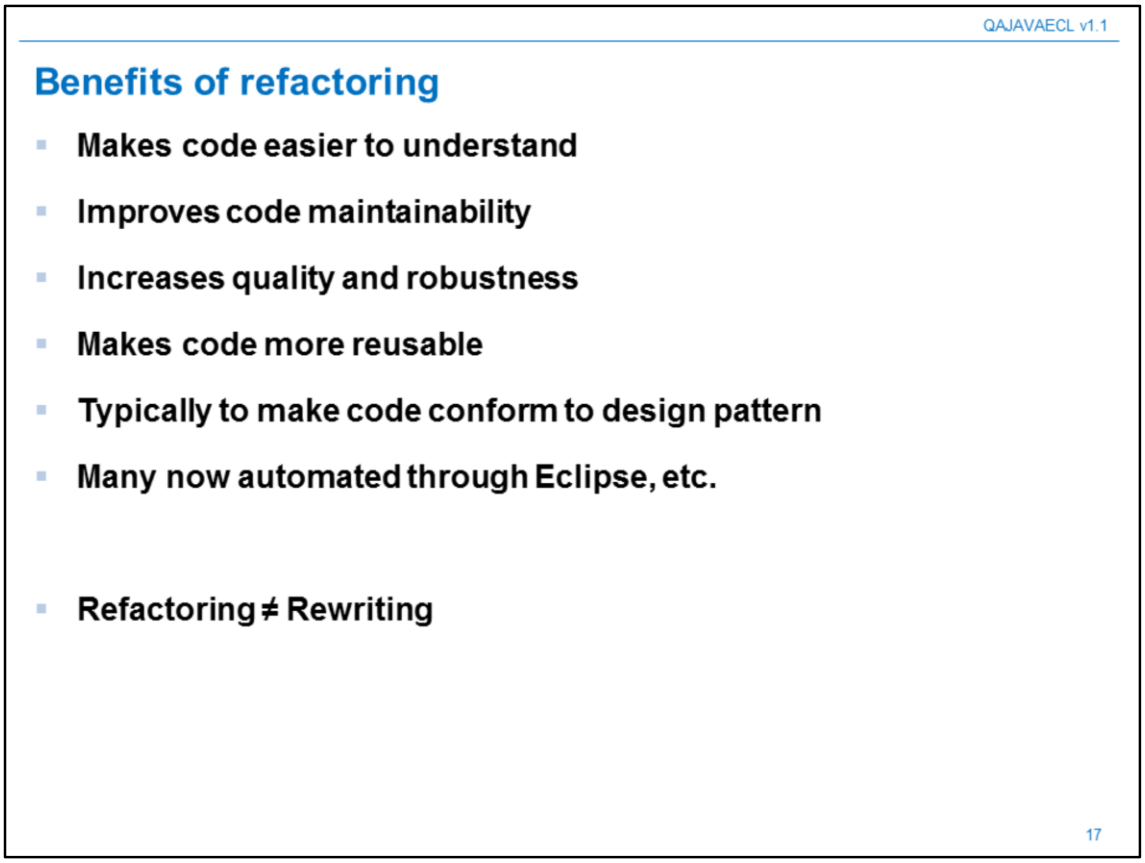

QAJAVAECL v1.1

Code smell

- **Code smell: symptom of something wrong with code**
  - Not *necessarily* a problem, but needs consideration
  - Typically an anti-pattern for a refactoring
- **Large class**
  - Class that has too much in it; aka God-like object
- **Feature envy**
  - A class that uses the methods of another class excessively
- **Inappropriate intimacy**
  - Class has dependencies on implementation details in another
- **Switch statement**
  - May point to better design using polymorphism

16

"Code smell" has become a common phrase in the XP/TDD community, promoted in Martin Fowler's Refactoring: Improving the Design of Existing Code. Here are just four representative examples. Switch statement illustrates that these are only indicative; a particular use of a switch statement may be a perfectly appropriate flow control construct.



QAJAVAECL v1.1

Benefits of refactoring

- Makes code easier to understand
- Improves code maintainability
- Increases quality and robustness
- Makes code more reusable
- Typically to make code conform to design pattern
- Many now automated through Eclipse, etc.

- Refactoring ≠ Rewriting

17

Koskela: "Do. Not. Skip. Refactoring

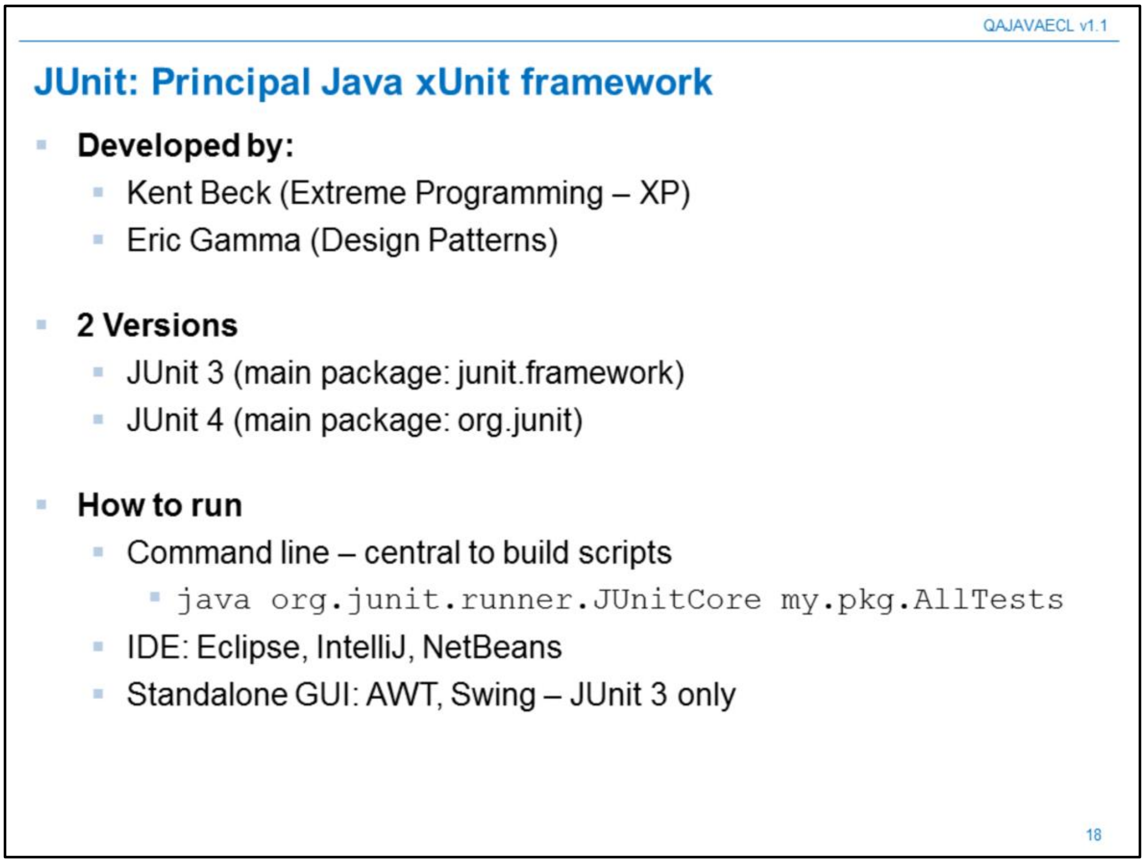
… The single biggest problem I've witnessed after watching dozens of teams take their first steps in test-driven development is insufficient refactoring." (Test Driven, p. 106)

Martin Fowler's site refactoring.com is a useful resource. See e.g. the catalogue of refactorings at:

http://www.refactoring.com/catalog/index.html

Another on-line catalogue of refactorings is available at:

http://industriallogic.com/xp/refactoring



QAJAVAECL v1.1

JUnit: Principal Java xUnit framework

- **Developed by:**
  - Kent Beck (Extreme Programming – XP)
  - Eric Gamma (Design Patterns)
- **2 Versions**
  - JUnit 3 (main package: junit.framework)
  - JUnit 4 (main package: org.junit)
- **How to run**
  - Command line – central to build scripts
    - `java org.junit.runner.JUnitCore my.pkg.AllTests`
  - IDE: Eclipse, IntelliJ, NetBeans
  - Standalone GUI: AWT, Swing – JUnit 3 only

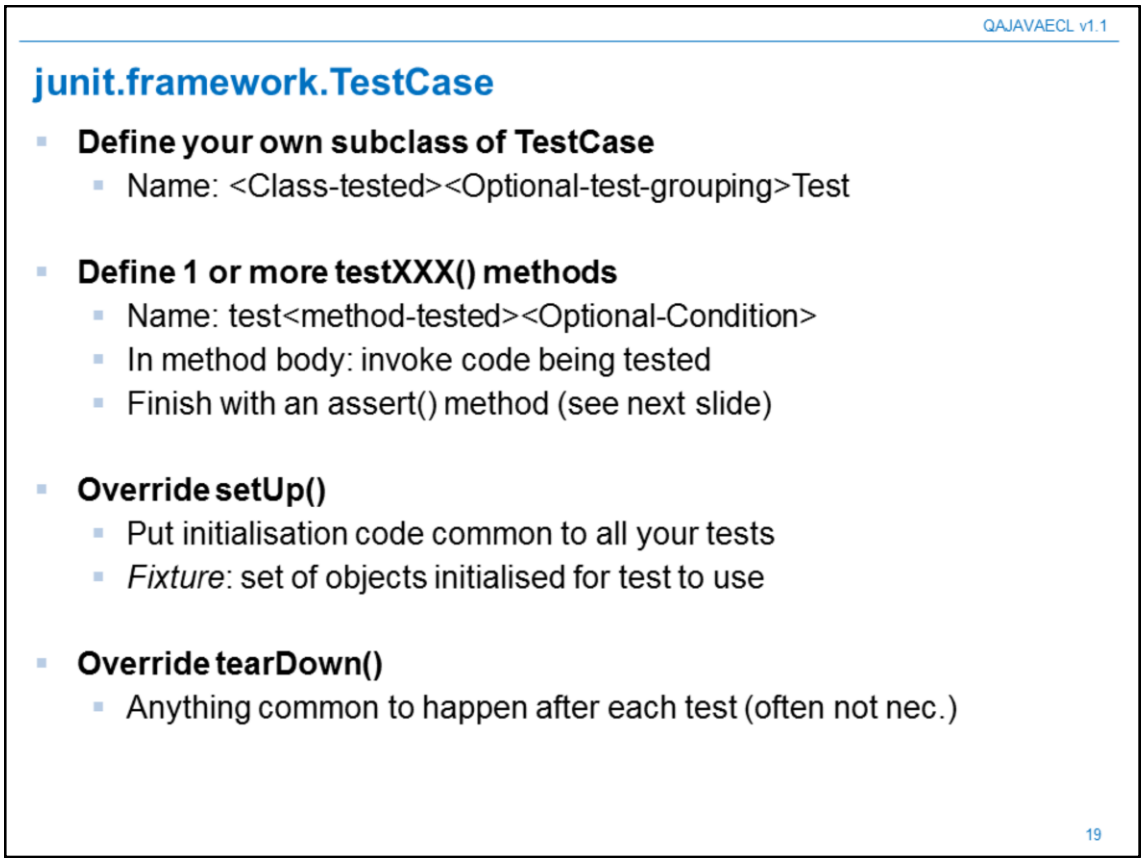
18

Another way the essence of the xUnit framework can be captured:

Arrange objects: create and set up as required

Act on an object: invoke the method being tested

Assert: what is expected



QAJAVAECL v1.1

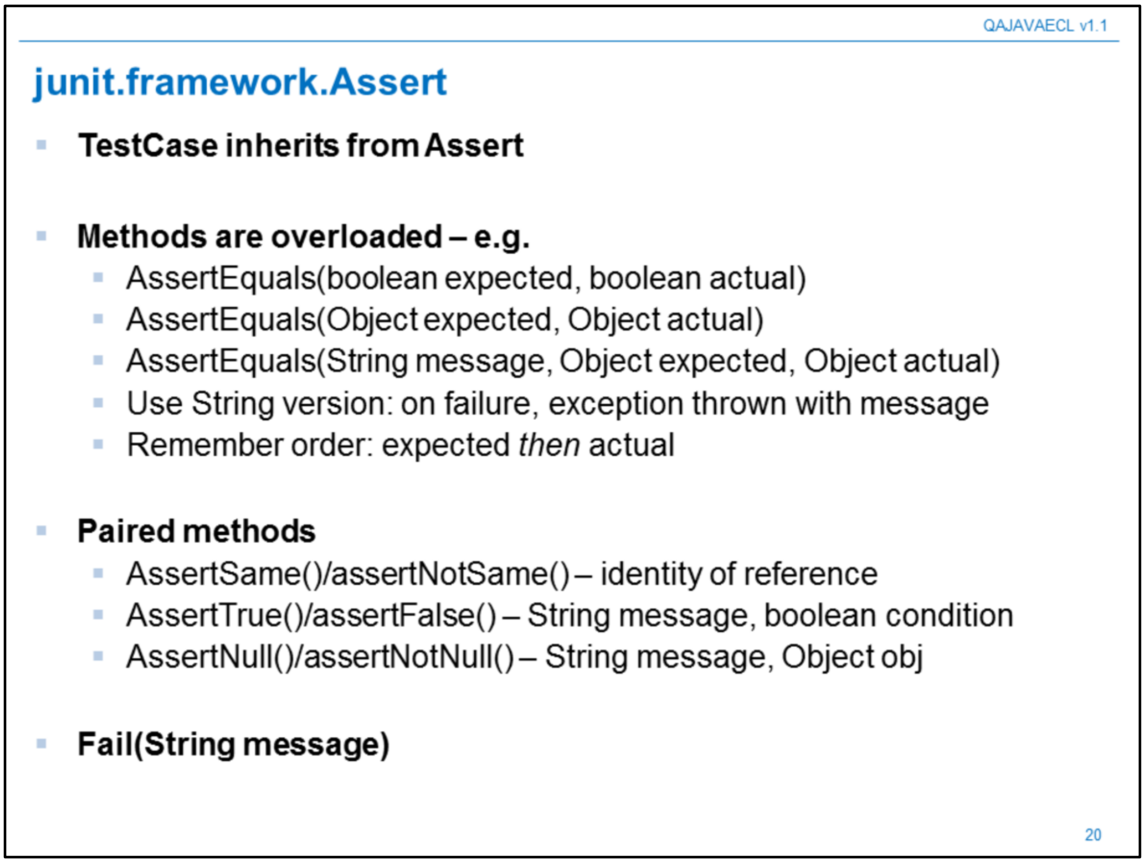
junit.framework.TestCase

- **Define your own subclass of TestCase**
  - Name: <Class-tested><Optional-test-grouping>Test
- **Define 1 or more testXXX() methods**
  - Name: test<method-tested><Optional-Condition>
  - In method body: invoke code being tested
  - Finish with an assert() method (see next slide)
- **Override setUp()**
  - Put initialisation code common to all your tests
  - *Fixture*: set of objects initialised for test to use
- **Override tearDown()**
  - Anything common to happen after each test (often not nec.)

19

It is general JUnit 3 best practice to always start a test method name 'test', and then to make the rest of the method name descriptive of what is being tested. (By default, JUnit 3 treats all methods starting 'test' as test methods, but it is not absolutely essential to follow this naming convention.)

A test's fixture is the set of background resources or data needed for it to run. Since several tests in the same test class can share this common initial state, fixture creation is put into the setUp() method, and destruction in the tearDown() method.



QAJAVAECL v1.1

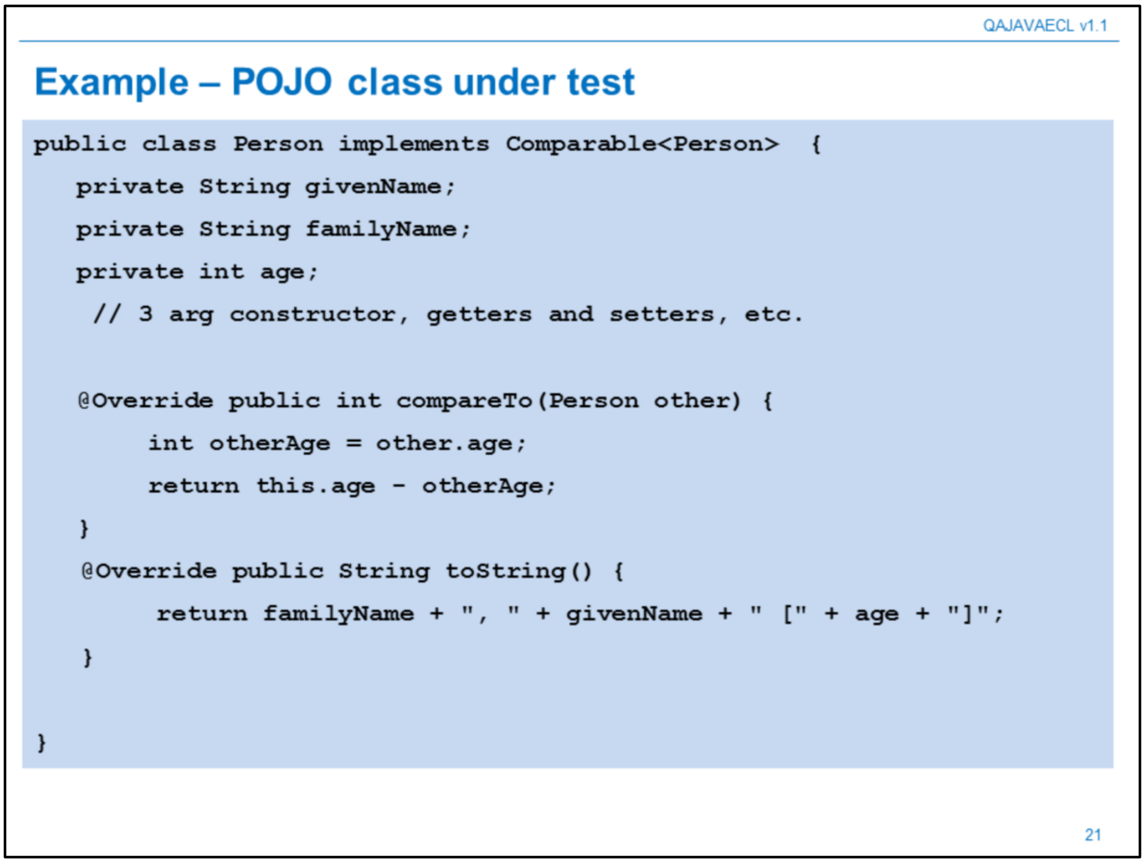
junit.framework.Assert

- **TestCase inherits from Assert**
- **Methods are overloaded – e.g.**
  - AssertEquals(boolean expected, boolean actual)
  - AssertEquals(Object expected, Object actual)
  - AssertEquals(String message, Object expected, Object actual)
  - Use String version: on failure, exception thrown with message
  - Remember order: expected *then* actual
- **Paired methods**
  - AssertSame()/assertNotSame() – identity of reference
  - AssertTrue()/assertFalse() – String message, boolean condition
  - AssertNull()/assertNotNull() – String message, Object obj
- **Fail(String message)**

20

assertEquals() is overloaded for all the primitives, and Object, and String – so 10 versions – then double that for the three argument version which starts with a String message. For the two items compared, always give the expected one first, then the computed one, for this is the order JUnit will use in error reporting (as in "java.lang.AssertionError: expected:<7.0> but was:<8.0>").

Always use the String version of an assert() method so that the reason for failure will be stated if the assert fails.



QAJAVAECL v1.1

Example – POJO class under test

```
public class Person implements Comparable<Person>  {
   private String givenName;
   private String familyName;
   private int age;
    // 3 arg constructor, getters and setters, etc.

   @Override public int compareTo(Person other) {
        int otherAge = other.age;
        return this.age - otherAge;
   }
   @Override public String toString() {
        return familyName + ", " + givenName + " [" + age + "]";
   }


}
```

21

A very common operation is performing a sort on a collection of objects. Typically, sort is implemented in terms of making pairwise comparisons of objects in the collection: to arrange a list in order we need to know how any two objects in the list compare. We need to define a comparison relation which will behave like this:

For an arbitrary pair of objects, for the notion of ordering we are defining:

The first comes before the second: return a negative number (e.g. –1)

The two are equivalent: return 0

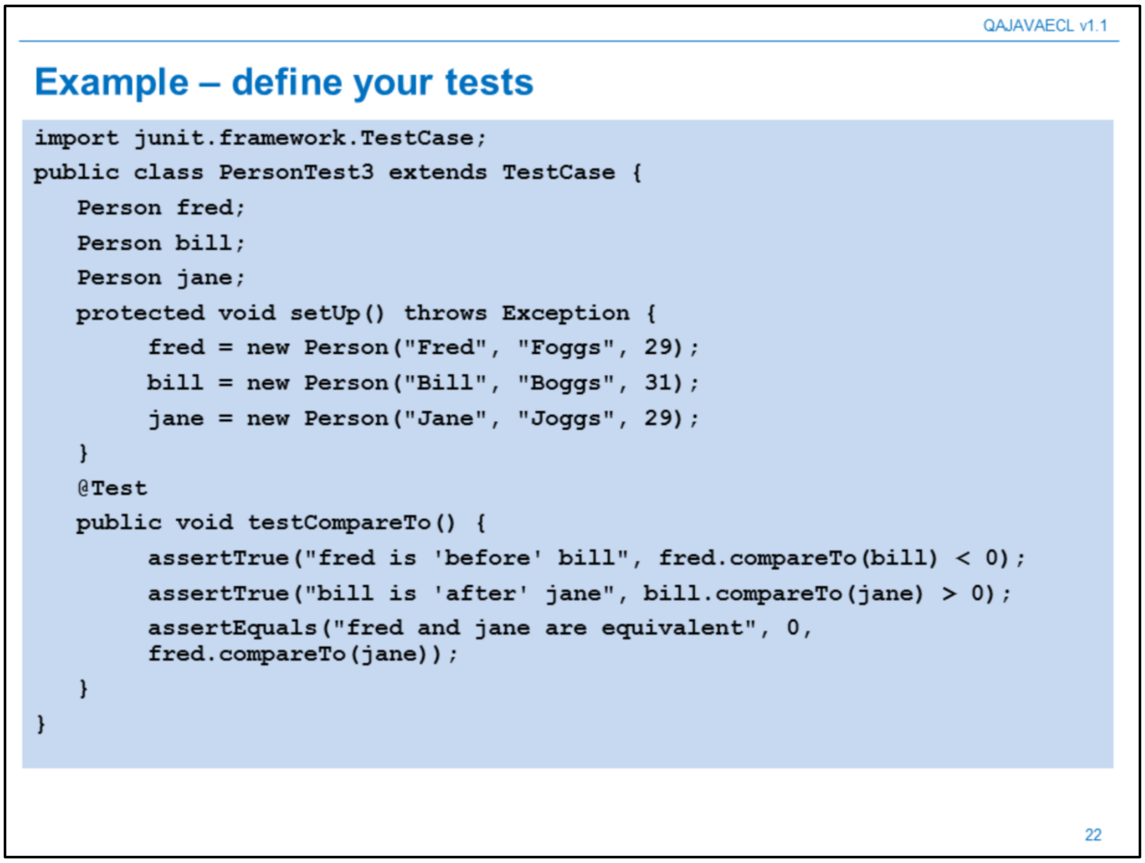
The first comes after the second: return a positive number (e.g. +1)

The Person class here implements Comparable – in other words, commits to defining an inherent notion of what it is for one Person to be before another if we were to sort them. We are building an inherent notion of order into the class, by implementing compareTo(). This definition is in terms of age:

If the other is older, return negative: this before the other

If the other is the same age, return 0: tied in ordering

If this older, return positive: this after the other


QAJAVAECL v1.1

Example – define your tests

```
import junit.framework.TestCase;
public class PersonTest3 extends TestCase {
   Person fred;
   Person bill;
   Person jane;
   protected void setUp() throws Exception {
        fred = new Person("Fred", "Foggs", 29);
        bill = new Person("Bill", "Boggs", 31);
        jane = new Person("Jane", "Joggs", 29);
   }
   @Test
   public void testCompareTo() {
        assertTrue("fred is 'before' bill", fred.compareTo(bill) < 0);
        assertTrue("bill is 'after' jane", bill.compareTo(jane) > 0);
        assertEquals("fred and jane are equivalent", 0,
        fred.compareTo(jane));
   }
}
```

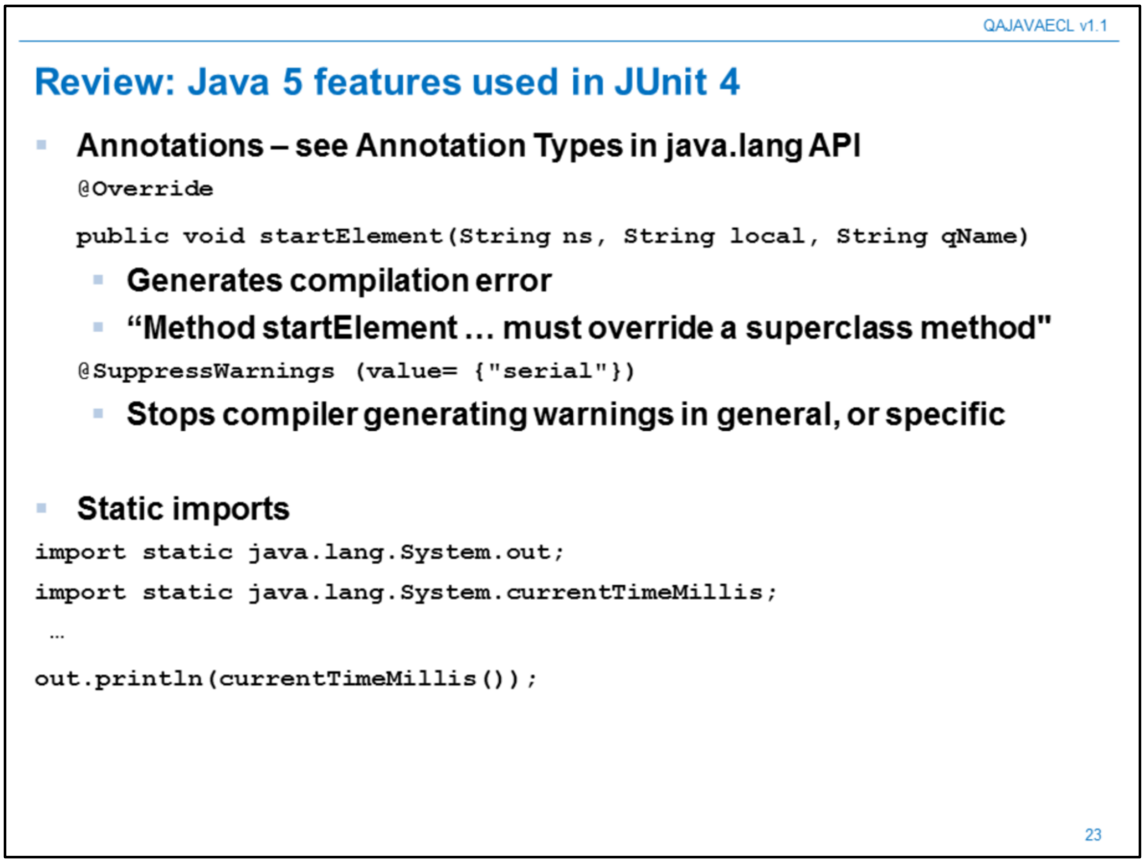
22

Here the emphasis is on testing the method – rather than a “behaviour” – and since there are three possible return values of compareTo(), it could be argued that it is natural to group together the three assertions that cover these three cases. However, that would be wrong.

For each test method:

There should be at least one assert (otherwise it’s not actually a test)

There should be at most one assert (if there is more than one assert, as soon as one fails, the remainder will not be exercised.)



QAJAVAECL v1.1

Review: Java 5 features used in JUnit 4

- Annotations – see Annotation Types in java.lang API

```
@Override
public void startElement(String ns, String local, String qName)
```

  - Generates compilation error
  - “Method startElement … must override a superclass method"

```
@SuppressWarnings (value= {"serial"})
```

  - Stops compiler generating warnings in general, or specific

- Static imports

```
import static java.lang.System.out;
import static java.lang.System.currentTimeMillis;
 …
out.println(currentTimeMillis());
```

23

The @Override example comes from the SAX approach to xml parsing. In SAX parsing, we define a handler class to receive events triggered by the SAX parser as it processes some xml – typically by subclassing org.xml.sax.helpers.DefaultHandler. This class defines a default, do-nothing implementation of the relevant listener/handler methods, in particular,

public void startElement(String ns, String local, String qName, Attributes atts)

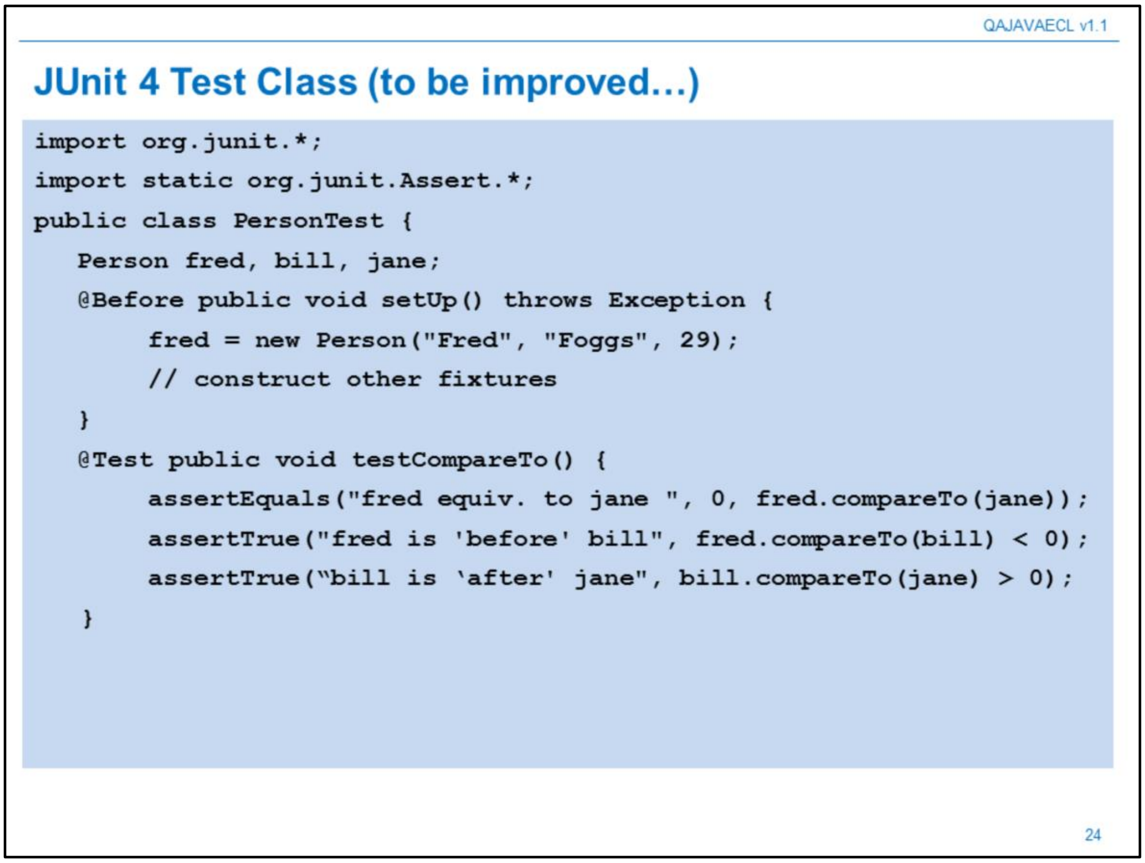
If you were to define your own handler subclass with a method with the signature shown on the slide,

public void startElement(String ns, String local, String qName)

everything would compile, the parser would run, but your handler would not be processing callbacks to the startElement method, because it had defined an irrelevant method with the wrong signature. By inserting the @Override annotation, we introduce a bit of meta-data into the code, and the compiler can use this to check if this method does indeed override one in the superclass.

The @SuppressWarnings example above illustrates the way that annotations can be qualified. This annotation has a 'value' element, which can be given a list of warning types to suppress. Or it can be used unqualified, to suppress any warnings.

With static imports, static members of a class – fields and methods – can be imported directly into another class, making the code cleaner.



QAJAVAECL v1.1

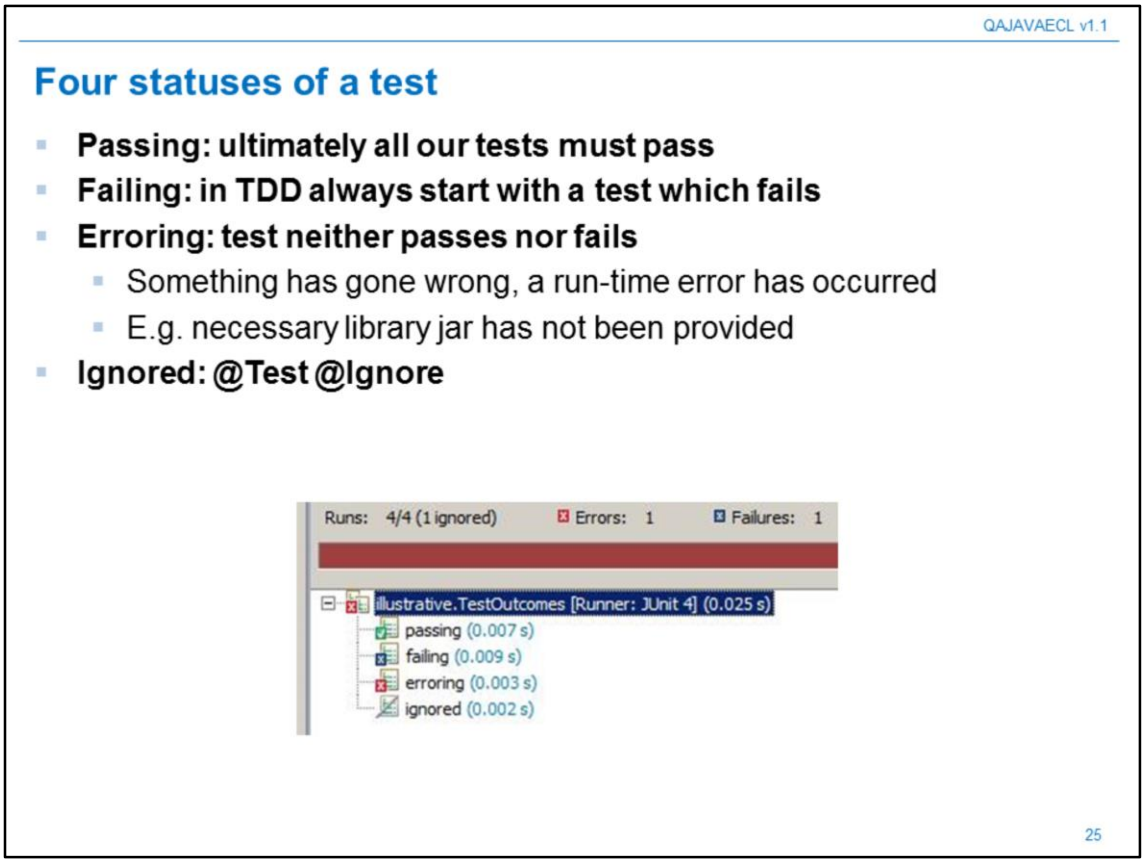
JUnit 4 Test Class (to be improved…)

```
import org.junit.*;
import static org.junit.Assert.*;
public class PersonTest {
   Person fred, bill, jane;
   @Before public void setUp() throws Exception {
        fred = new Person("Fred", "Foggs", 29);
        // construct other fixtures
   }
   @Test public void testCompareTo() {
        assertEquals("fred equiv. to jane ", 0, fred.compareTo(jane));
        assertTrue("fred is 'before' bill", fred.compareTo(bill) < 0);
        assertTrue("bill is 'after' jane", bill.compareTo(jane) > 0);
   }
```

24

JUnit 4 defines its own custom annotations, and we must import them to use them. (Our test class no longer subclasses one in the JUnit API.) It is now the annotations @Before, @Test, @After that mark methods as respectively setUp, test and tearDown methods, so those methods could conform to different naming conventions.

If multiple classes have common setUp()/tearDown() code, abstract this out to a common superclass. Don't put any @Test methods in this superclass (else they will be inherited in each sub-class and therefore run multiple times). The common @Before method will be called before any @Before method in a subclass. Naturally, if the method is called both setUp() in sub and super class, the former will override the latter. A class can have multiple @Before methods (say setUp() and init()), but apart from this case of inheritance, it would not be a good design to have two set-up methods defined in the same class.



QAJAVAECL v1.1

Four statuses of a test

- **Passing: ultimately all our tests must pass**
- **Failing: in TDD always start with a test which fails**
- **Erroring: test neither passes nor fails**
  - Something has gone wrong, a run-time error has occurred
  - E.g. necessary library jar has not been provided
- **Ignored: @Test @Ignore**

```
Runs:  4/4 (1 ignored)     Errors:  1     Failures:  1

illustrative.TestOutcomes [Runner: JUnit 4] (0.025 s)
    passing (0.007 s)
    failing (0.009 s)
    erroring (0.003 s)
    ignored (0.002 s)
```

25

The @Ignore annotation is more useful than it might first seem. If you know you have to write a test, but can't get round to writing it just yet, you can add this annotation alongside @Test. The essential point is that you get a constant reminder, as shown in the screen-shot above, that there is a test there, that it needs tending to.

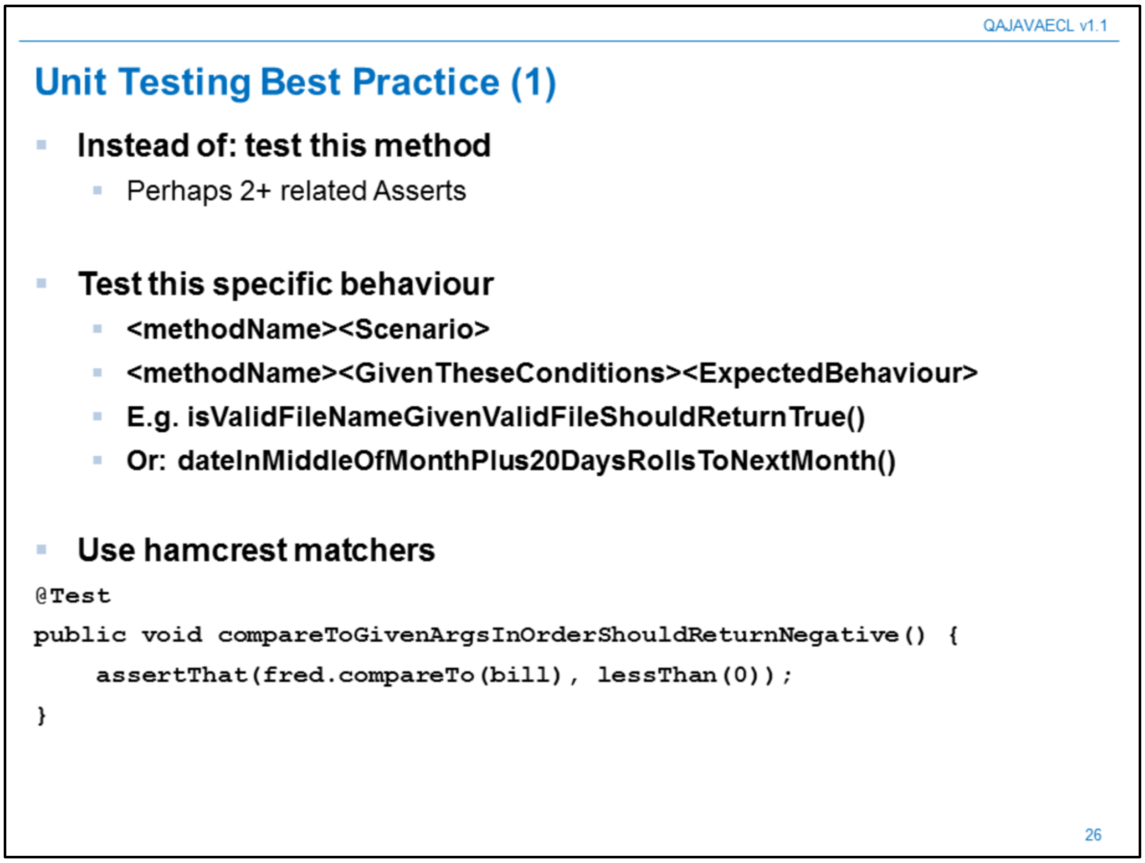
If you left off @Ignore and just had an empty test body, the test would trivially pass, and you might easily overlook it

If you left off @Ignore and put a fail() in as Eclipse does, that wouldn't be right either

If you commented out @Test, the method would be there in your test class but it wouldn't show up in your test summary, so again you might forget about it

Note that the @Ignore annotation can be qualified with a string reason, as in @Ignore("WiP"), etc. Under no circumstances do this kind of thing:

```
@Test
public void testXyzMethod() {
    // TODO: how do we test this?
    assertTrue(true);
}
```



QAJAVAECL v1.1

Unit Testing Best Practice (1)

- **Instead of: test this method**
  - Perhaps 2+ related Asserts
- **Test this specific behaviour**
  - **<methodName><Scenario>**
  - **<methodName><GivenTheseConditions><ExpectedBehaviour>**
  - **E.g. isValidFileNameGivenValidFileShouldReturnTrue()**
  - **Or: dateInMiddleOfMonthPlus20DaysRollsToNextMonth()**
- **Use hamcrest matchers**

```
@Test
public void compareToGivenArgsInOrderShouldReturnNegative() {
    assertThat(fred.compareTo(bill), lessThan(0));
}
```

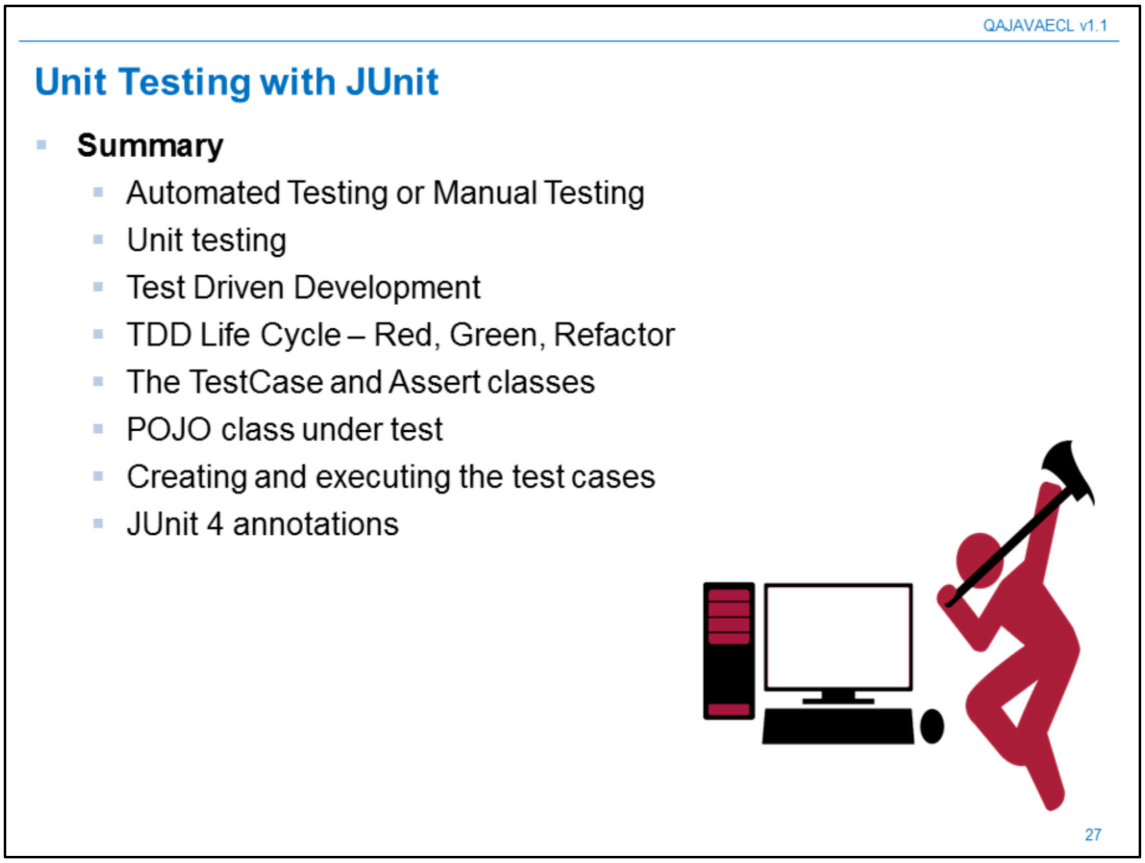
26

JUnit 4 encourages a subtle but important shift of emphasis in the way we conceive of tests. The naming convention here follows Osherove's proposal The Art of Unit Testing, p. 29. Whatever naming style you follow, what's important is that test method names can't be too long.

One popular style is to structure tests into a Given/When/Then format. Bob Martin (Clean Code, Ch. 9) gives this example of breaking up a long test method:

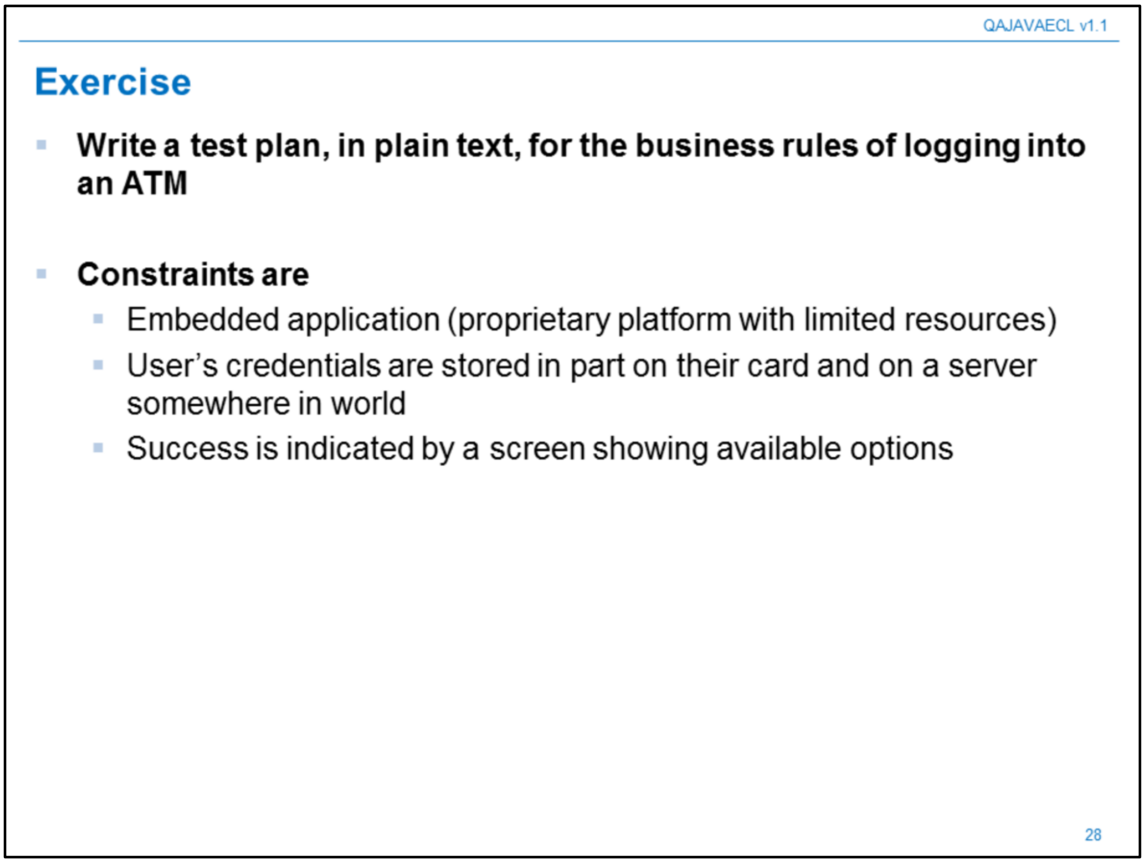
```
@Test public void getPageHierarchyAsXml() {
   givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
   whenRequestIsIssued("root", "type:pages");
   thenResponseShouldBeXml();
}
```

Notice the last bulleted example on the slide does not even mention the method being exercised: it describes the scenario being played out in that test.



Unit Testing with JUnit

- **Summary**
  - Automated Testing or Manual Testing
  - Unit testing
  - Test Driven Development
  - TDD Life Cycle – Red, Green, Refactor
  - The TestCase and Assert classes
  - POJO class under test
  - Creating and executing the test cases
  - JUnit 4 annotations

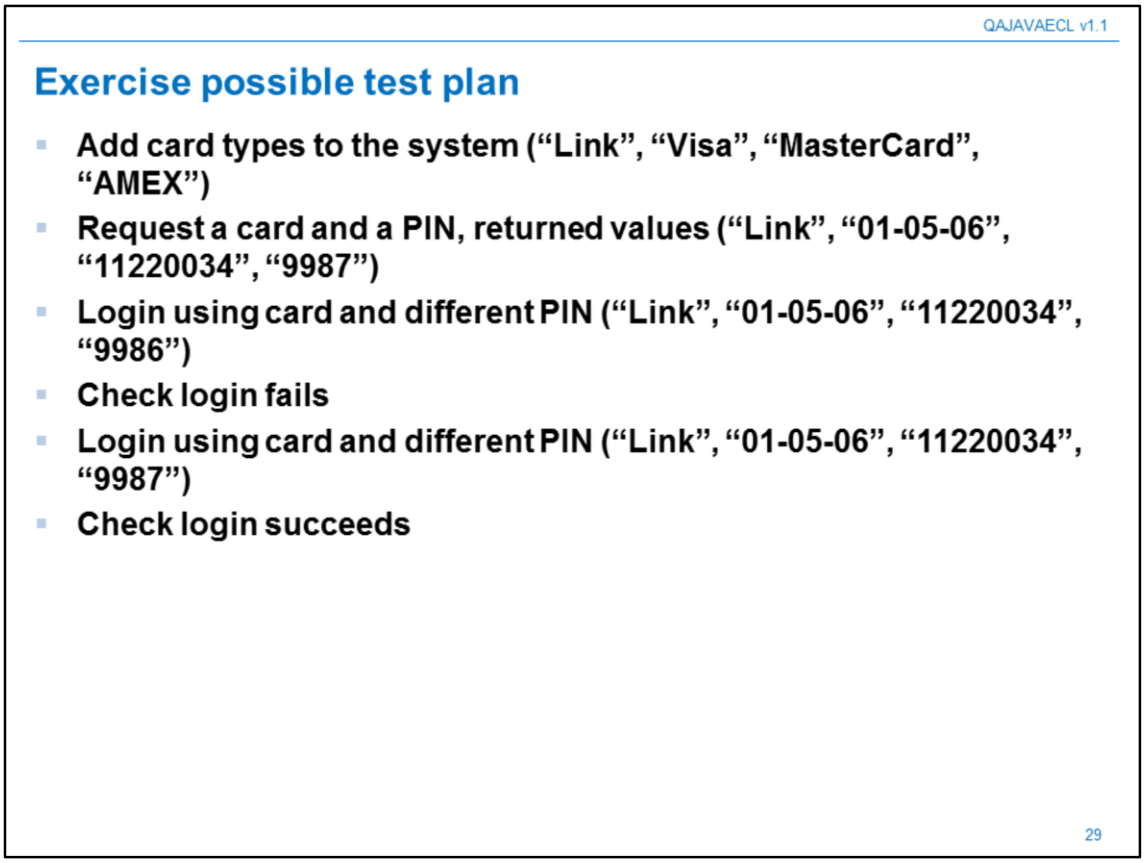


QAJAVAECL v1.1

Exercise

- **Write a test plan, in plain text, for the business rules of logging into an ATM**
- **Constraints are**
  - Embedded application (proprietary platform with limited resources)
  - User's credentials are stored in part on their card and on a server somewhere in world
  - Success is indicated by a screen showing available options

28



QAJAVAECL v1.1

Exercise possible test plan

- Add card types to the system ("Link", "Visa", "MasterCard", "AMEX")
- Request a card and a PIN, returned values ("Link", "01-05-06", "11220034", "9987")
- Login using card and different PIN ("Link", "01-05-06", "11220034", "9986")
- Check login fails
- Login using card and different PIN ("Link", "01-05-06", "11220034", "9987")
- Check login succeeds

29