

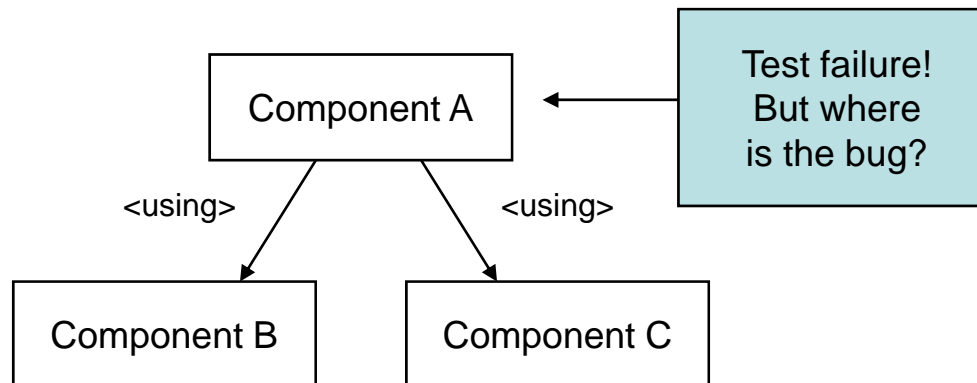
Unit Testing

and

JUnit

# Problem area

- Code components must be tested!
  - Confirms that your code works
- Components must be tested in isolation
  - A functional test can tell you that a bug exists in the implementation
  - A unit test tells you where the bug is located



# Example: The Calculator

```
public interface Calculator
{
    int add( int number1, int number2 );

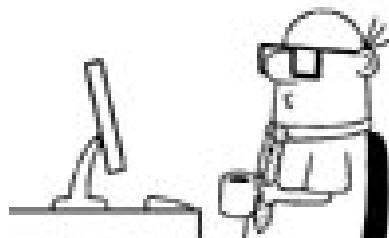
    int multiply( int number1, int number2 );
}
```

```
public class DefaultCalculator
    implements Calculator
{
    public int add( int number1, int number2 )
    {
        return number1 + number2;
    }

    public int multiply( int number1, int number2 )
    {
        return number1 * number2;
    }
}
```

# Approaches to unit testing

- Write a small command-line program, enter values, and verify output
  - Involves your ability to type numbers
  - Requires skills in mental calculation
  - Doesn't verify your code when its released



# Approaches to unit testing

- Write a simple test program
  - Objective and preserves testing efforts
  - Requires you to monitor the screen for error messages
  - Inflexible when more tests are needed

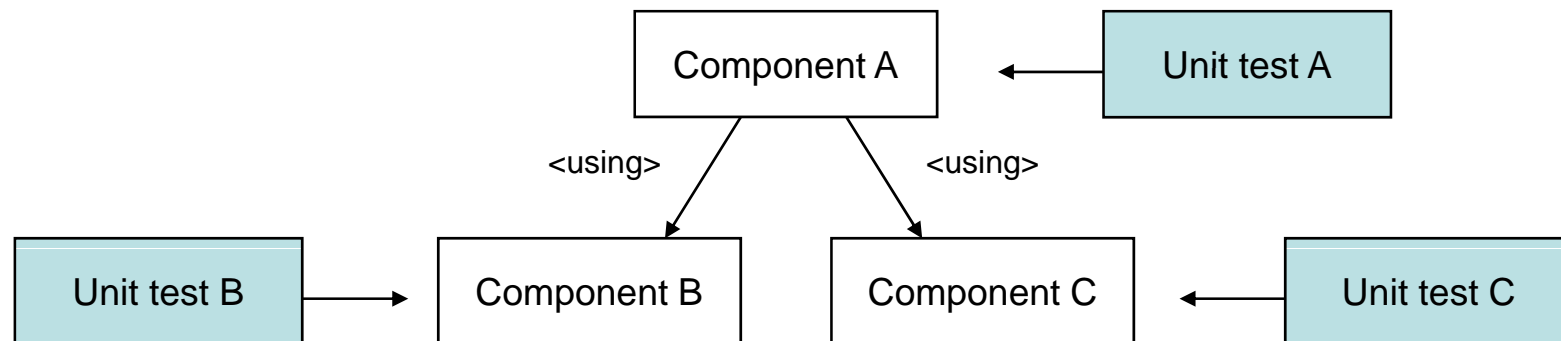
```
public class TestCalculator
{
    public static void main( String[] args )
    {
        Calculator calculator = new DefaultCalculator();

        int result = calculator.add( 8, 7 );

        if ( result != 15 )
        {
            System.out.println( "Wrong result: " + result );
        }
    }
}
```

# The preferred solution

- Use a unit testing framework like *JUnit*
- A *unit* is the smallest testable component in an application
- A unit is in most cases a *method*
- A unit does not depend on other components which are *not unit tested themselves*
- Focus on whether a method is following its *API contract*



# JUnit

- *De facto* standard for developing unit tests in Java
  - One of the most important Java libraries ever developed
  - Made unit testing easy and popular among developers
- Two techniques:
  - Extending the TestCase class (prior to version 4)
  - Using Java annotations (after version 4)

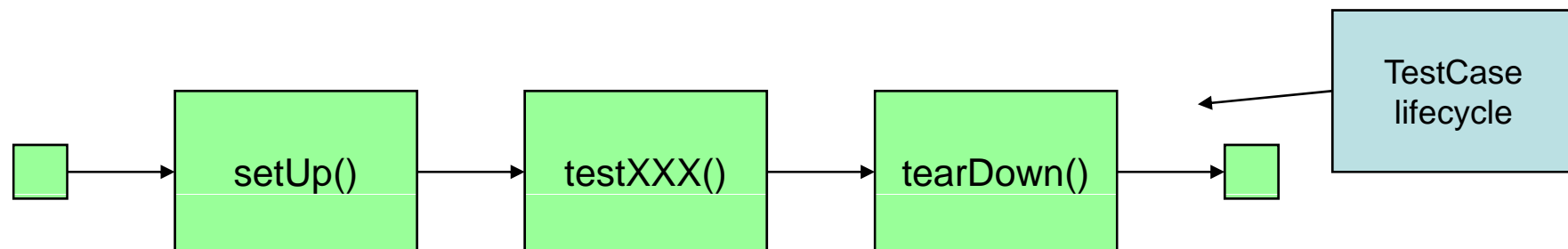
# Extending the TestCase class

- Your test class should *extend* the TestCase class
- Will find and execute all methods starting with *test* in your test class
- Lets you set up a test *fixture* by overriding the *setUp* and *tearDown* methods
- Provides methods for verifying method output through the *Assert* class



# Test fixtures

- Tests may require common resources to be set up
  - Complex data structures
  - Database connections
- A *fixture* is a set of common needed resources
- Common setup code inside tests doesn't make sense
- A fixture can be created by overriding the *setUp* and *tearDown* methods from `TestCase`
- *setUp* is invoked before each test, *tearDown* after



# JUnit Calculator test

Import TestCase



Extend TestCase



Override setUp()



Make methods that begin with *test*



Use assertEquals to verify output



```
import junit.framework.TestCase;

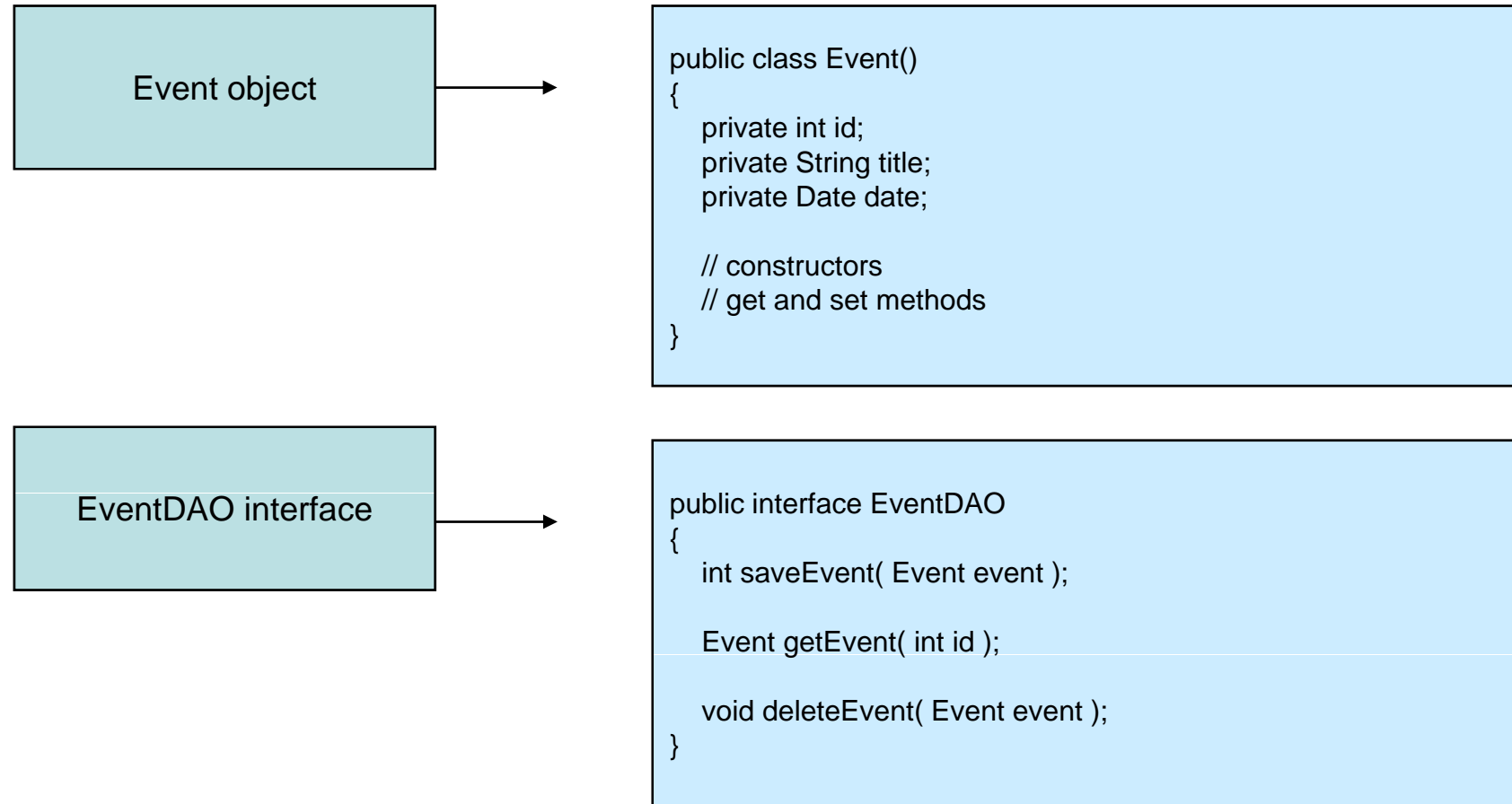
public class CalculatorTest
    extends TestCase
{
    Calculator calculator;

    public void setUp()
    {
        calculator = new DefaultCalculator();
    }

    public void testAdd()
    {
        int sum = calculator.add( 8, 7 );

        assertEquals( sum, 15 );
    }
}
```

# Example: The EventDAO



# Fixture in EventDAOTest

Overrides setUp(), will be invoked before each test

Instantiates an EventDAO, Date, and Event for use in test methods

Using the EventDAO and Event inside a test

```
public class EventDAOTest
    extends TestCase
    {
        private EventDAO eventDAO;
        private Date date;
        private Event event;

        public void setUp()
        {
            eventDAO = new MemoryEventDAO();

            Calendar calendar = Calendar.getInstance();
            calendar.set( 2007, Calendar.NOVEMBER, 10 );
            date = calendar.getTime();

            event = new Event( "U2 concert", date );
        }

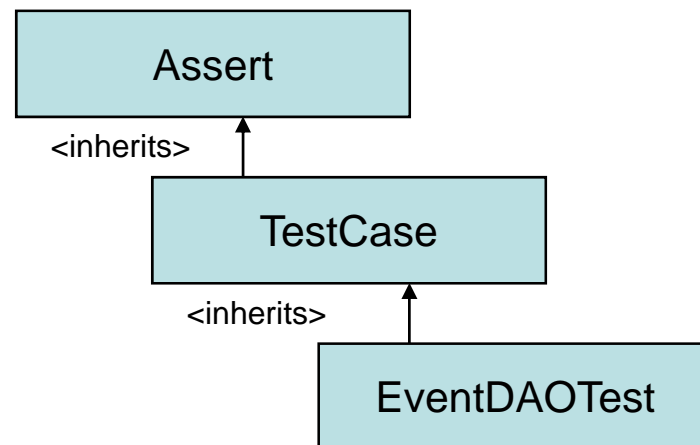
        public void testAddEvent()
        {
            int id = eventDAO.saveEvent( event );

            // more testing code follows
        }

        // more test methods
    }
}
```

# The Assert class

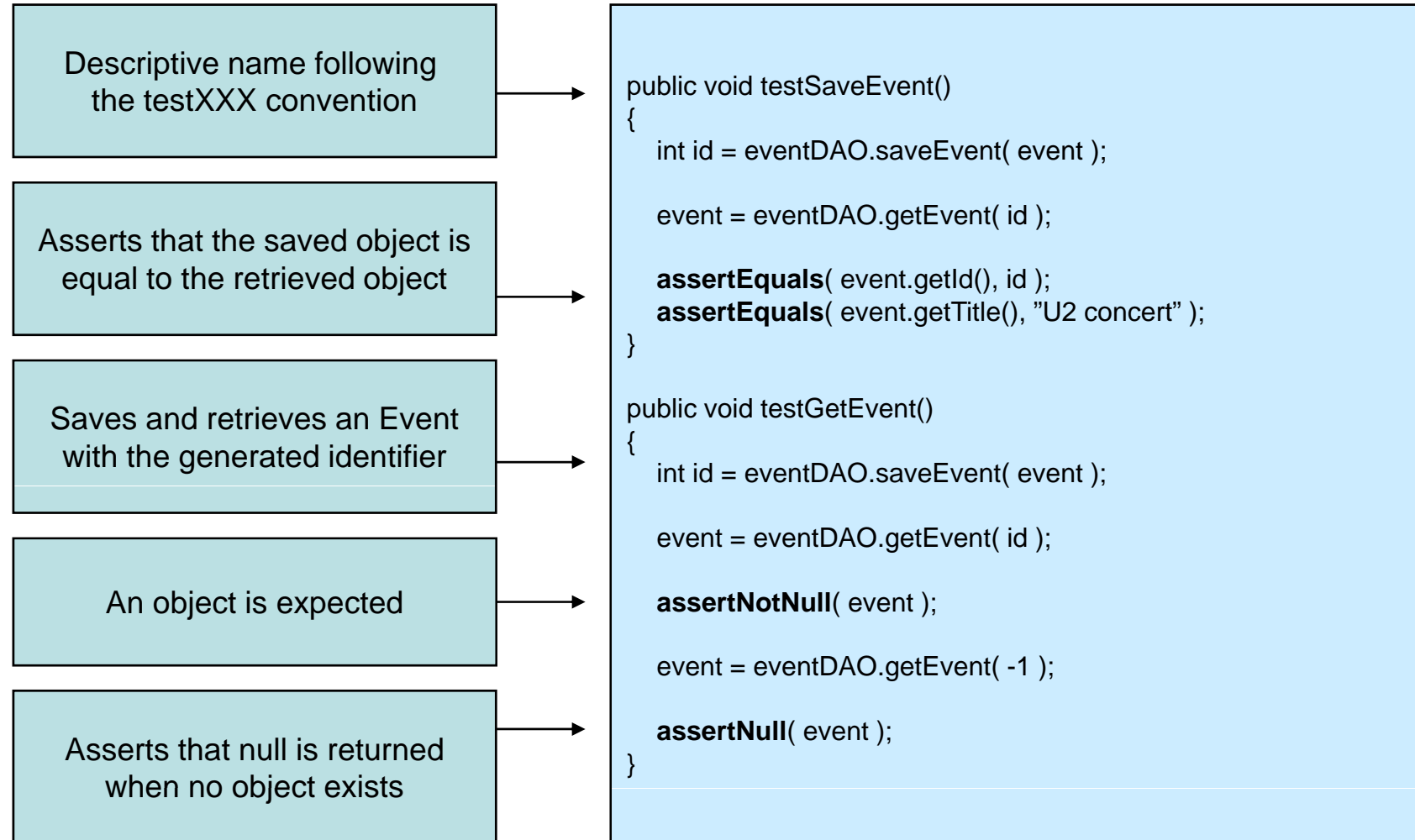
- Contains methods for testing whether
  - Conditions are true or false
  - Objects are equal or not
  - Objects are null or not
- If the test fails, an `AssertionFailedError` is thrown
- All methods have overloads for various parameter types
- Methods available because *TestCase* inherits *Assert*



# Assert methods

<b><i>Method</i></b>	<b><i>Description</i></b>
assertTrue( boolean )	Asserts that a condition is true.
assertFalse( boolean )	Asserts that a condition is false.
assertEquals( Object, Object )	Asserts that two objects are equal.
assertNotNull( Object )	Asserts that an object is <i>not</i> null.
assertNull( Object )	Asserts that an object is null.
assertSame( Object, Object )	Asserts that two references refer to the same object.
assertNotSame( Object, Object )	Asserts that two references do <i>not</i> refer to the same object.
fail( String )	Asserts that a test fails, and prints the given message.

# Assert in EventDAOTest

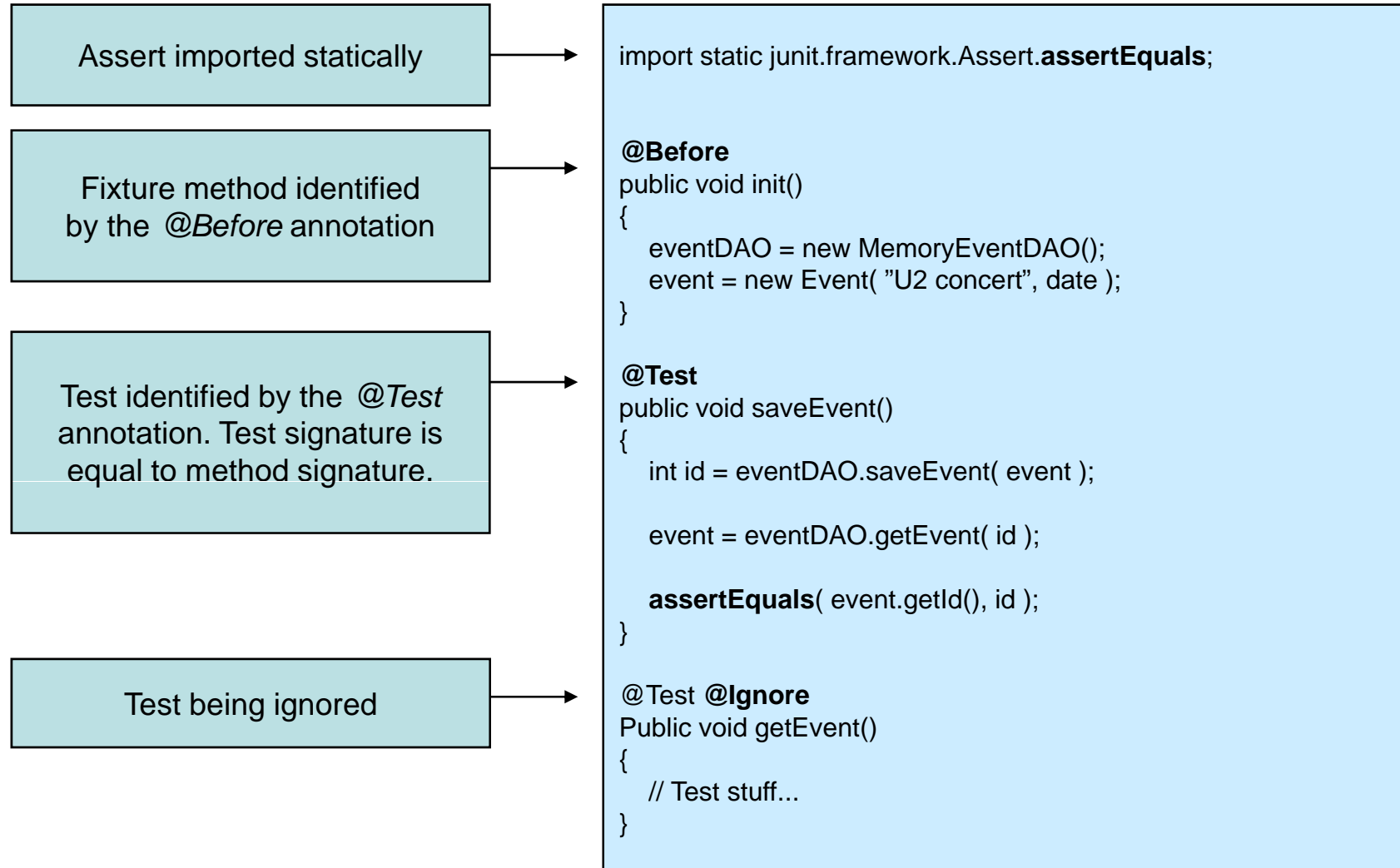


# Using Java annotations

- No need to follow JUnit naming conventions
  - Tests identified by the `@Test` annotation
  - Fixture methods identified by `@Before` and `@After` annotations
- Class-scoped fixture
  - Identified by the `@BeforeClass` and `@AfterClass` annotations
  - Useful for setting up expensive resources, but be careful...
- Ignored tests
  - Identified by the `@Ignore` annotation
  - Useful for slow tests and tests failing for reasons beyond you
- Timed tests
  - Identified by providing a parameter `@Test( timeout=500 )`
  - Useful for benchmarking and network testing



# EventDAOTest with annotations



# Testing Exceptions

- Methods may be required to throw exceptions
- Expected exception can be declared as an annotation
  - `@Test( expected = UnsupportedOperationException.class )`

Annotation declares that an exception of class `UnsupportedOperationException` is supposed to be thrown



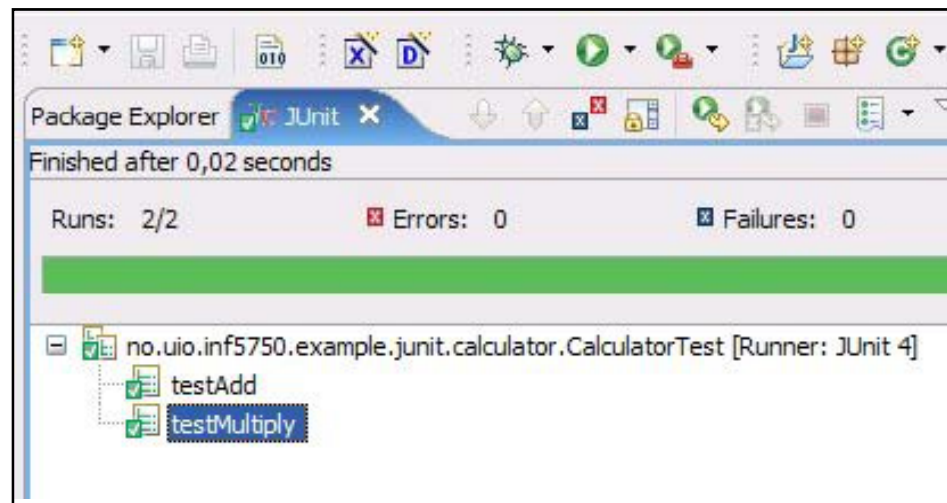
```
@Test( expected = UnsupportedOperationException.class )  
public void divideByZero()  
{  
    calculator.divide( 4, 0 );  
}
```

# Running JUnit

- Textual test runner
  - Used from the command line
  - Easy to run
- Integrate with Eclipse
  - Convenient, integrated testing within your development environment!
- Integrate with Maven
  - Gets included in the build lifecycle!

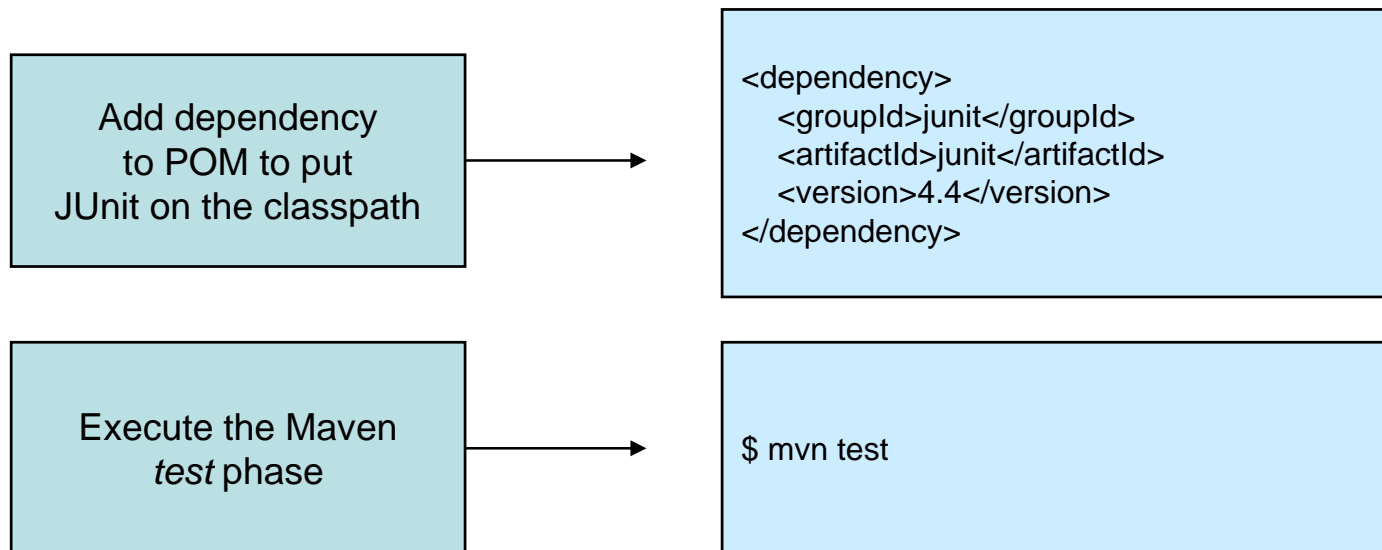
# JUnit with Eclipse

- Eclipse features a JUnit view
- Provides an informativ GUI displaying test summaries
- Lets you edit the code, compile and test without leaving the Eclipse environment



# JUnit with Maven

- Maven provides support for automated unit testing through JUnit
- Unit testing is included in the build lifecycle
  - Verifies that existing components work when other components are added or changed



# JUnit with Maven

- Maven requires all test-class names to contain *Test*
- Standard directory for test classes is `src/test/java`
  - Maven will execute all tests for you
- The *test* phase is mapped to the *Surefire* plugin
- Surefire will generate reports based on your test runs
- Reports are located in *target/surefire-reports*

```
-----
T E S T S
-----
Running no.uio.inf5750.example.junit.calculator.CalculatorTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.09 sec
Running no.uio.inf5750.example.junit.event.EventDAOTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.03 sec
Results :
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 2 seconds
[INFO] Finished at: Sun Sep 16 13:40:58 CEST 2007
[INFO] Final Memory: 3M/127M
[INFO] -----
```

# Best practises

- One unit test for each tested method
  - Makes debugging easier
  - Easier to maintain
- Choose descriptive test method names
  - TestCase: Use the testXXX naming convention
  - Annotations: Use the method signature of the tested method
- Automate your test execution
  - If you add or change features, the old ones must still work
  - Also called *regression testing*
- Test more than the "happy path"
  - Out-of-domain values
  - Boundary conditions

# Advantages of unit testing

- Improves debugging
  - Easy to track down bugs
- Facilitates refactoring
  - Verifies that existing features still work while changing the code structure
- Enables teamwork
  - Lets you deliver tested components without waiting for whole application to finish
- Promotes object oriented design
  - Requires your code to be divided in small, re-usable units
- Serving as developer documentation
  - Unit tests are samples that demonstrates usage of the API



# Resources

- Vincent Massol: *JUnit in Action*
  - Two free sample chapters
  - <http://www.manning.com/massol>
- JUnit home page
  - Articles and forum
  - <http://www.junit.org>
- Articles
  - <http://www-128.ibm.com/developerworks/java/library/j-junit4.html>
  - <http://www-128.ibm.com/developerworks/opensource/library/os-junit/>