Danmarkskort: Visualisering, Navigation, Søgning og Ruteplanlægning

Lecture 3: Black box testing and JUnit

Troels Bjerre Sørensen

Today's lecture

- Black-box testing
  - Shortcomings of white box testing
  - Coverage criteria
  - Examples

- JUnit
  - Installation
  - API
  - Examples

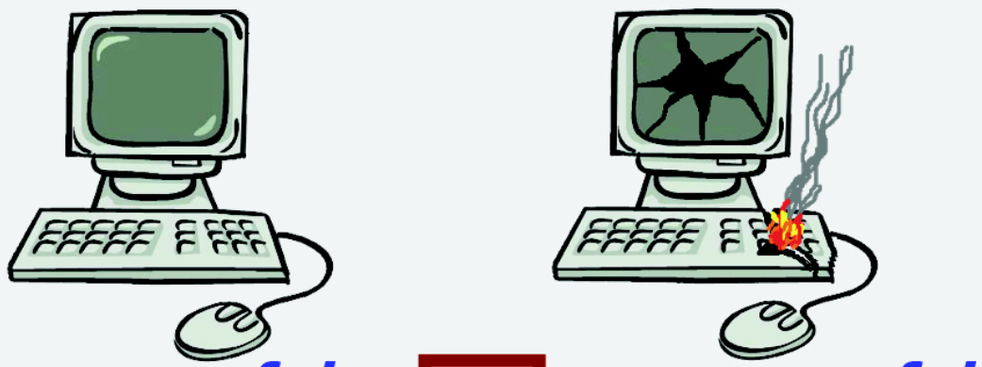
Psychology of Testing

[cf. ”The Art of Software Testing” (Chap. 2), Glenford J. Myers, 1979]

Definition (Glen J. Myers, ”The art of software testing). Testing is the process of executing a program with the intent of finding errors.

- Goal: find as many errors as possible

Problematic terminology:



vs.

Much better terminology:

Constructive thinking

- Test to pass
- Testing own code

vs.

Destructive thinking

- Test to fail
- Testing someone else’s code

Test suite

[cf. ”The Art of Software Testing” (Chap. 2), Glenford J. Myers, 1979]

Definition (Glen J. Myers, ”The art of software testing). Testing is the process of executing a program with the intent of finding errors.

- Test case: an input to the program, and an expected output
- Test suite: the collections of all tests associated with a project
- Test suite is an expectancy table:

Test ID	Input	Expected output
A	(98, 3)	”a=108, r=6”
B	(76, 4)	”a=86, r=2”

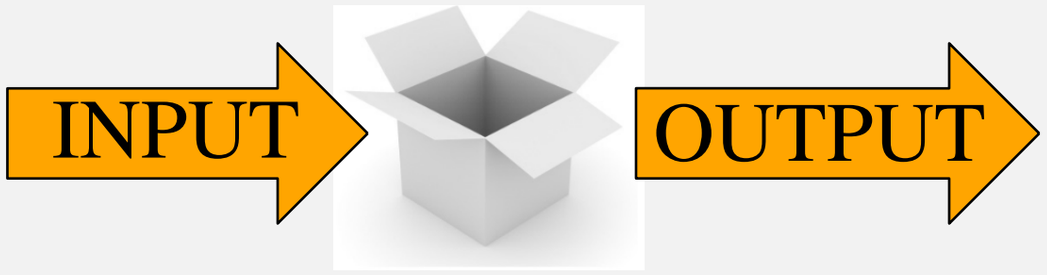
```
void m(int account, int rate) {
    account = account + 10;

    if (account > 90)    /* 1 */
        rate = rate * 2;
    else
        rate = rate / 2;

    out(“account = ” + account
        + “ rate = ” + rate);
}
```

Complementary approaches to test suite construction

White box testing

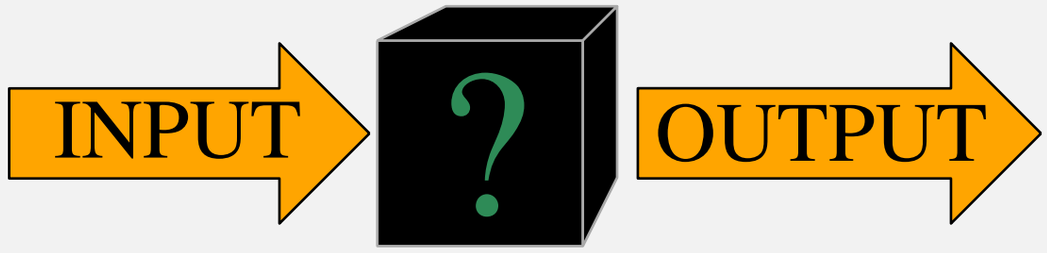


- Generate test input from Source Code:

```
x = in();
if (odd(x)) {
   x = x/2;
} else {
   x = 3*x+1;
}
out(x);
```

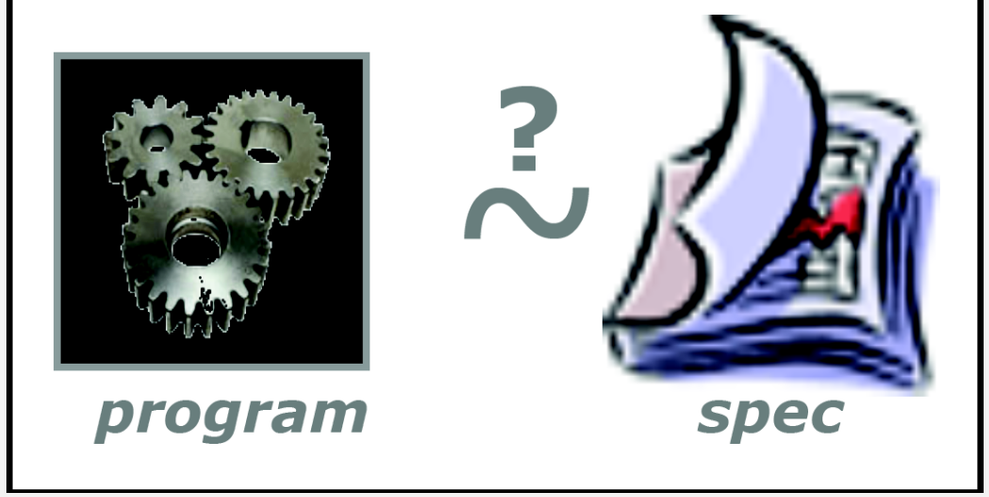

- ... then check specs for expected behavior

Black box testing



- Generate test cases from specs

> The program should compute a single step of the Collatz conjecture function

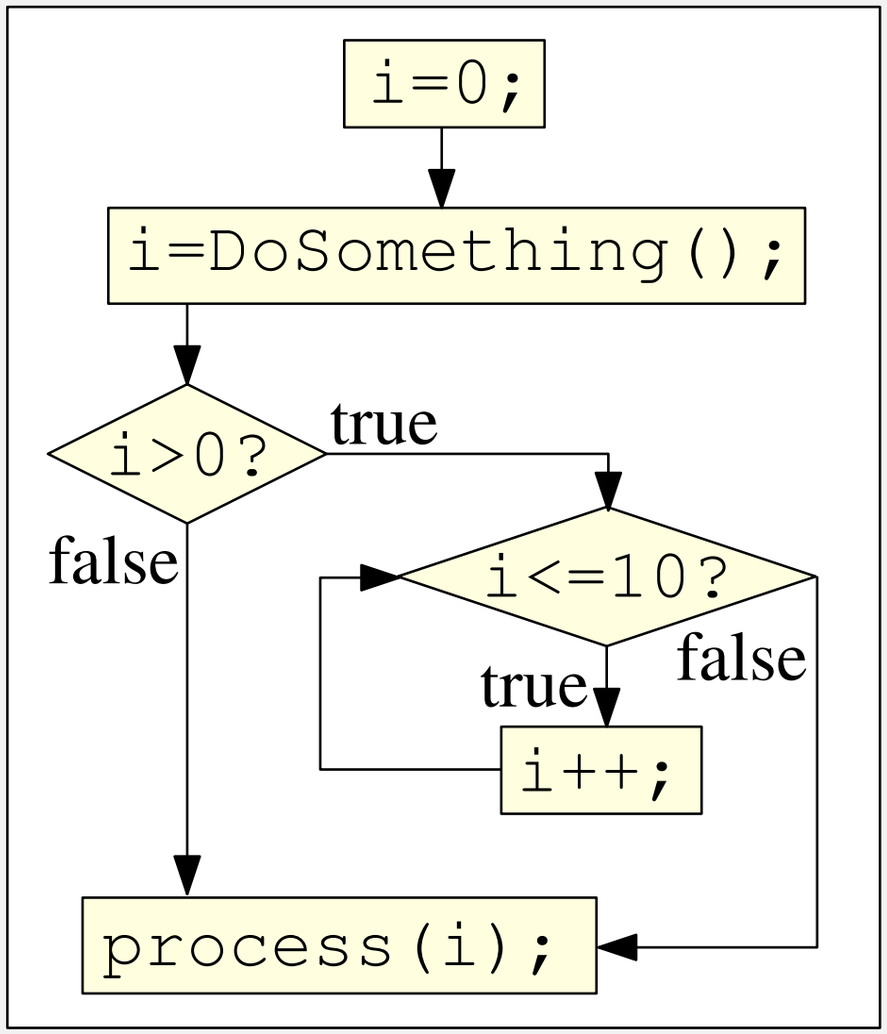


Systematic testing

Different criteria for coverage:

- **Method coverage:**
  Every method is tested
  (at least once)
- **Statement coverage:**
  Every statement is executed
  (at least once)
- **Branch coverage:**
  All branches in control flow
  are executed (at least once)
- **Path coverage:**
  All paths are executed
  (at least once)

```
int i = 0;
i = DoSomething();
if (i>0){
    while (i<=10)
          i++;
}
process(i);
```



Coverage table

- A white box provides a coverage table to ensure coverage

```
void m(int account, int rate) {
    account = account + 10;

    if (account > 90)    /* 1 */
        rate = rate * 2;
    else
        rate = rate / 2;

    out("account = " + account
        + " rate = " + rate);
}
```

Coverage table:

Branch	Input property	Covered by test
(1) `true`	`account > 90`	A
(1) `false`	`account <= 90`	B

Expectancy table:

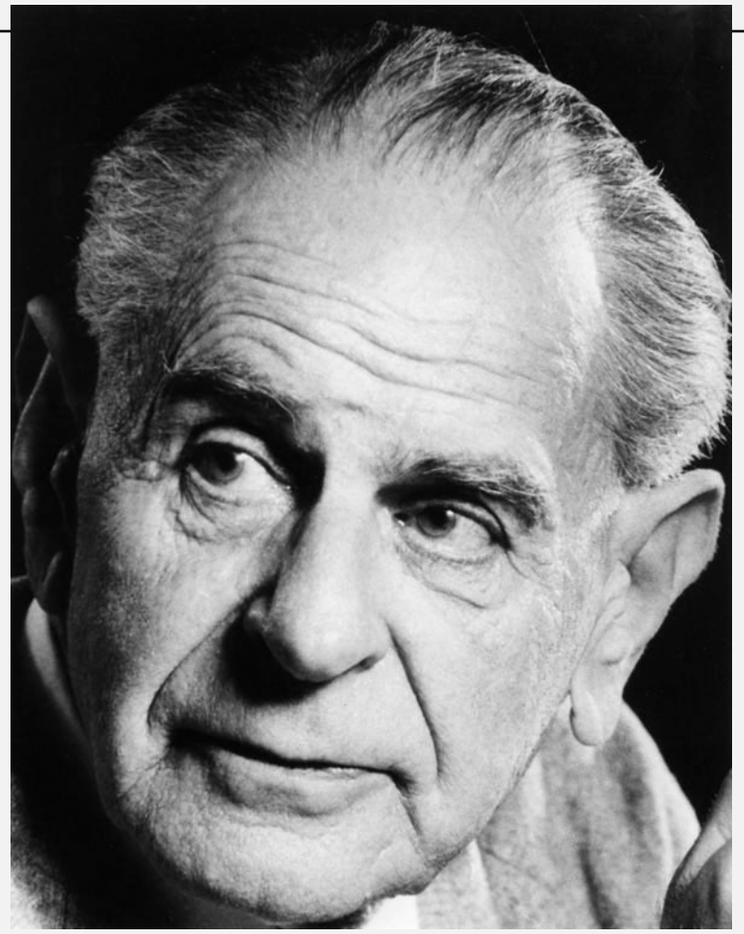
Test ID	Input	Expected output	Actual output	Outcome
A	(98, 3)	”a=108, r=6”	”a=108, r=6”	pass
B	(76, 4)	”a=86, r=2”	”a=86, r=2”	pass

Path coverage: still finite

Statement	Cases to test
`if`	Condition false and true
`for`	Zero, one, and more than one iterations
`while`	Zero, one, and more than one iterations
`do-while`	One, and more than one, iterations
`switch`	Every `case` and `default` branch must be executed
`try-catch-finally`	The `try` clause, every `catch` clause, and the `finally` clause must be executed

Summary

- We can't test every input
- Tests cannot prove the absence of bugs, but proper coverage can heighten our confidence in the correctness of the program
- White box testing uses source code to generate test input
- A test case takes a path through the program
- Coverage criteria:
  - Method coverage:
    each method is reached by some path
  - Statement coverage:
    each node in the control graph is used by some path
  - Branch coverage:
    each edge in the control graph is used by some path
  - Path coverage:
    all paths are included in the suite



“My proposal is based upon an asymmetry between verifiability and falsifiability; an asymmetry which results from the logical form of universal statements. For these are never derivable from singular statements, but can be contradicted by singular statements.”

Karl Popper (1902-1994)
The Logic of Scientific Discovery

Is this method correct?

```
int next_prime(int p) {
    return (p == 2 ? 3 : p+2);
}
```

- Full path coverage:

Branch	Input	Expected output	Actual output
p==2	2	3	3
p!=2	3	5	5
p!=2	5	7	7

Is this program correct?

```
int f(int x) {
    int a = 0;
    while (x > 0) {
        a += x;
        x--;
    }
    return a;
}
```

- It depends on the specs:

> "The function f must compute the sum of the set of integers bounded by its argument."

- Specs can be unclear, or even contain bugs itself

Reading the specs carefully

- Positive universals:
  - ’Always’, ’for every’, ’for each’, ’for all’, . . .
  - Look for exceptions.

- Negative universals:
  - ’None’, ’never’, ‘under no circumstances’, . . .
  - Look for exceptions.

- Hybris:
  - ’Trivially’, ’certainly’, ’clearly’, ’obviously’, ’evidently’, . . .
  - Well, really?

Reading the specs carefully (cont.)

- Unspecified conditionals:
  - ’Some(-times)’, ’often’, ’usually’, ‘ordinarily’, ’mostly’, . . .
  - Ask: Exactly when? And if not?

- Continuations:
  - ’Etcetera’, ’and so forth’, ’and so on’, . . .
  - Ask: What is the implicit generalisation?

- Examples:
  - ’E.g.’, ’for example’, ’such as’, . . .
  - Ask: Is it representative?

Reading the specs carefully (cont.)

- Positive adjectives:
  - 'Good', 'fast', 'efficient', 'small', 'reliable', 'stable',...
  - Ask: How do these quantify?

- Allegedly completed:
  - 'Handled', 'processed', 'taken care of', 'eliminated'
  - Ask: Nothing is hidden, left out?

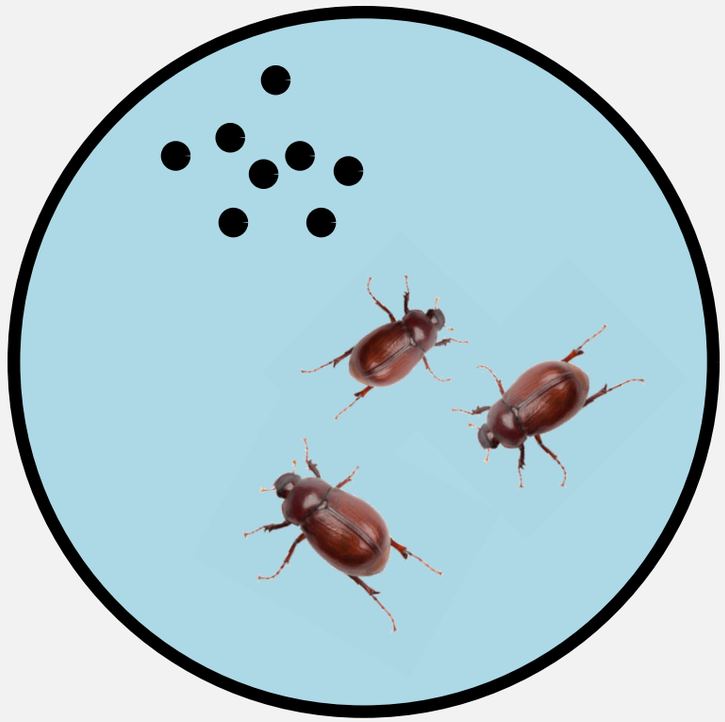
- Incompleted:
  - 'Skipped', 'unnecessary', 'superfluous', 'rejected'
  - Ask: Nothing is hidden, left out?

- Finally, watch out for:
  - "If ... Then" (with missing "Else"):
  - Ask: Well, what if not?

Fade to black

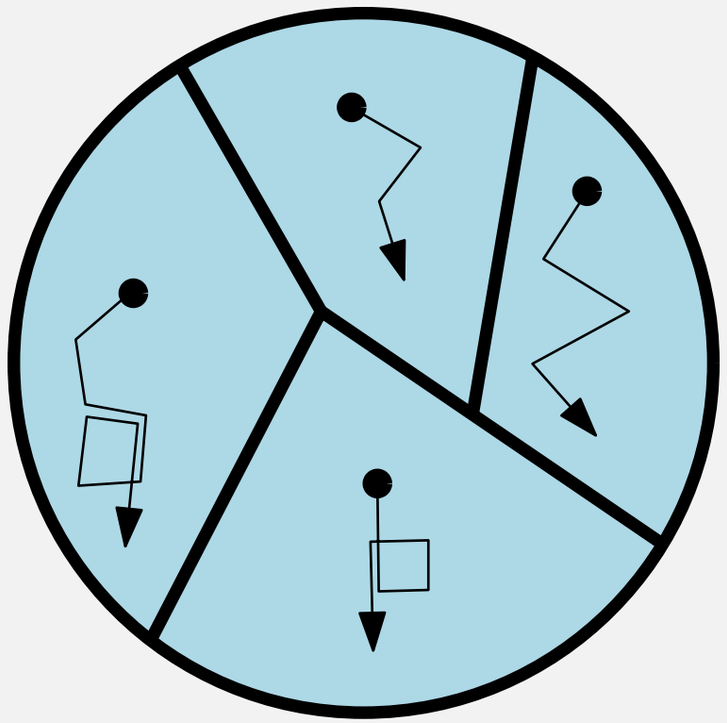
- We still cannot test the infinitely many possible inputs

- A big test suite wont help,
  if it does not cover properly



Fade to black

- We still cannot test the infinitely many possible inputs
- A big test suite wont help,
  if it does not cover properly
- White box test got coverage from
  arguing about paths each test case took

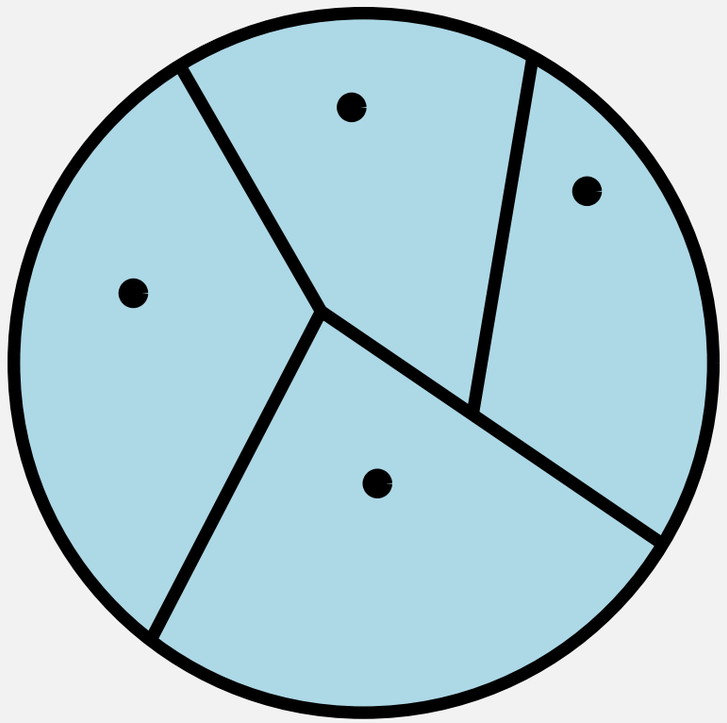


- Finitely many **types** of paths + one test case per type of path

Fade to black

- We still cannot test the infinitely many possible inputs

- A big test suite wont help,
  if it does not cover properly



- White box test got coverage from
  arguing about paths each test case took

- Finitely many **types** of paths + one test case per type of path

- Black box test: from specs, derive **equivalence classes** of inputs

Example 1

Given a (possibly empty) sequence of numbers, find the smallest and the greatest of these numbers.

Input property
No numbers
One number
Two numbers, equal
Two numbers, increasing
Two numbers, decreasing
Three numbers, increasing
Three numbers, decreasing
Three numbers, greatest in the middle
Three numbers, smallest in the middle

- Assumption:
  Same case $\Rightarrow$ Same error and Same error $\Rightarrow$ Same case

Example 1 (cont.)

Coverage table:

Input property	Input data set
No numbers	A
One number	B
Two numbers, equal	C1
Two numbers, increasing	C2
Two numbers, decreasing	C3
Three numbers, increasing	D1
Three numbers, decreasing	D2
Three numbers, greatest in the middle	D3
Three numbers, smallest in the middle	D4

Expectancy table:

Input data set	Contents	Expected output	Actual output
A	(no numbers)	Error message	
B	17	17 17	
C1	27 27	27 27	
C2	35 36	35 36	
C3	46 45	45 46	
D1	53 55 57	53 57	
D2	67 65 63	63 67	
D3	73 77 75	73 77	
D4	89 83 85	83 89	

Example 2

Given a (possibly empty) sequence of numbers, find the greatest difference between two consecutive numbers.

Input property
No numbers
One number
Two numbers, equal
Two numbers, increasing
Two numbers, decreasing
Three numbers, increasing difference
Three numbers, decreasing difference

Example 2 (cont.)

Coverage table:

Input property	Input data set
No numbers	A
One number	B
Two numbers, equal	C1
Two numbers, increasing	C2
Two numbers, decreasing	C3
Three numbers, increasing difference	D1
Three numbers, decreasing difference	D2

Expectancy table:

Input data set	Contents	Expected output	Actual output
A	(no numbers)	Error message	
B	17	Error message	
C1	27 27	0	
C2	36 37	1	
C3	48 46	2	
D1	57 56 59	3	
D2	69 65 67	4	

Common way to derive equivalence classes

- For each input (incl. state of object):
  - Split into a finite number of classes, e.g.:
    * `int`: negative, zero, positive
    * `String`: null, empty string, single character, long string
    * `List`: increasing, decreasing, unordered
    * ...
- Equivalence classes for whole = all combinations of classes for parts
- Bugs often occur around boundaries

Property	Input
Pos, Pos	(1,2)
Neg, Pos	(-3,4)
Zero,Pos	(0,5)
Pos, Neg	(6,-7)
Neg, Neg	(-8,-9)
Zero,Neg	(0,-10)
Pos, Zero	(11,0)
Neg, Zero	(-12,0)
Zero,Zero	(0,0)

Rules of thumb

- Avoid expected output is zero
- Automate tests
- Make sure each test case serves a purpose
- Try to guess what could go wrong:
  - Compare $<$ instead of $\leq$
  - References assumed to be not null
  - Index not negative
  - ...
- Good coverage: combine approaches
  - Construct black box tests from specs
  - Write the code
  - Check white box coverage, and add needed test cases

[Heavy inspiration from Bogdan Petrutescu slides, 2013]

Why JUnit

- "JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks." [http://junit.sourceforge.net]

- Open source, very popular

- Extensions

- Standard tool in development testing

- Generators for JUnit in almost any IDE (NetBeans, Eclipse..)

Features

- Assertions (result testing)
- Fixtures
- Suites
- Loop protection with timeouts
- Exception handling
- … much more

Example

```
package tests;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

/**
 * @author Bogdan Petrutescu
 */
public class StringSimpleTst {

   private String myString;

   @Before
   public void setUp() {
         this.myString = new String("First-year Project: Map of Denmark");
   }

   @Test
   public void testContainingString() {
         assertTrue("Denmark is present in the string",
                     myString.contains("Denmark"));
```

@Before tells JUnit that the following method must be run before each test case

@Test is an annotation, telling JUnit that the following method is a test case

Assertions are statements about what we expect to be true for correct code.

Assertions

Assertion	Semantic
assertEquals(Object exp, Object act)	exp.equals(act)
assertSame(Object exp, Object act)	exp == act
assertTrue(boolean cond)	cond is true
assertNull(Object o)	o == Null
assertArrayEquals(Object[] exp, Object[] act)	|exp|=|act| ^ (exp[i] = act[i])
assertThat(T e, Matcher<T> m)	m.matches(e)
fail()	fail the test case

- A test can contain as many assertions as needed
- If any assertion fails, the test case fails and aborts

Annotations

Annotation	Description
@Test	Test case. It identifies that a method is a test method.
@Before	Test "constructor". It executes the method before each test. Useful to prepare the test environment: e.g. read data, initialize the class.
@After	Test "deconstructor". It executes the method after each test. Useful to cleanup the test environment: e.g. delete temporary data.
@BeforeClass	Test class "constructor". It executes the method only once, before all tests start. Useful to perform time expensive processes (e.g. connecting to a database). All methods annotated with this annotation need be static to work with JUnit.
@AfterClass	Test class "deconstructor". It executes the method only once, after all tests are finished. Useful to perform clean-up (e.g. disconnecting from a database). All methods annotated with this annotation need to have a static modifier.
@Ignore	Test case. It ignores the test method. Useful when source code has changed and the test case was not updated or when the execution time is too long. It shouldn't be used without a very good reason.

Timeout and exceptions

- Optional parameters for @Test:

@Test parameters	Description
@Test (expected = Exception.class)	Fails the current JUnit test if the method does not throw the named exception or if it throws a different exception than the one declared.
@Test(timeout=N)	Fails the current JUnit test if the method takes longer than N milliseconds.

- Can be combined:

```
@Test(timeout=1000,expected=IndexOutOfBoundsException.class)
public void testDodgyCode() {
    ...
```

Suites

- Suites lets you combine many test classes into one:

  - **Example: creating a suite with all the tests from test classes MyClassTestA and MyClassTestB**

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

@RunWith(Suite.class)
@Suite.SuiteClasses({ MyClassTestA.class , MyClassTestB.class })
public class AllTests {...}
```

- Easy way to manage which tests are run
- Very useful for IDE integration

DEMO