Testing with JUnit

Object-Oriented Programming Lecture 7

IJP (Liang): chapter 44 (online material)

https://junit.org/junit4/

March 09, 2021

Radboud University

UNIT TESTING

the need for testing

the compiler checks

- syntax, types, some binding errors
- a program that is fine for the compiler can still do something entirely wrong

software failures are expensive

- annual costs of software failure in EU + USA: G€ 200 (€ 200 000 000 000, 200 billion €)
- annual costs failed software projects NL: G€ 2 (2 billion €)
- about 60% of the problems are 'simple'

testing is by far the most used way to detect errors

- testing takes up to 50% of project costs

Automated/systematic testing makes it better and cheaper

- a program to test programs
- JUnit is the standard testing tool for Java

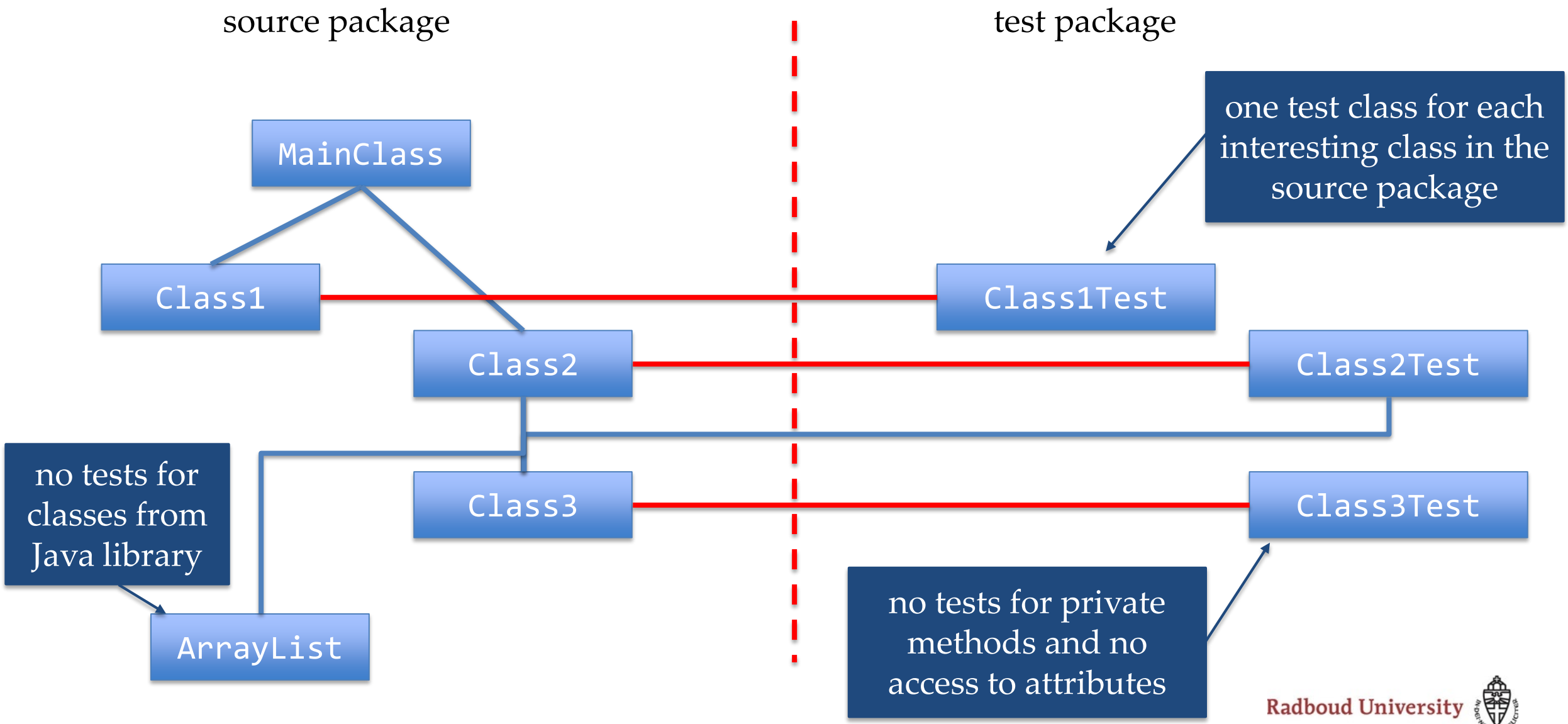
requirements for test system

tests are separated from ordinary code
tests can be executed automatically

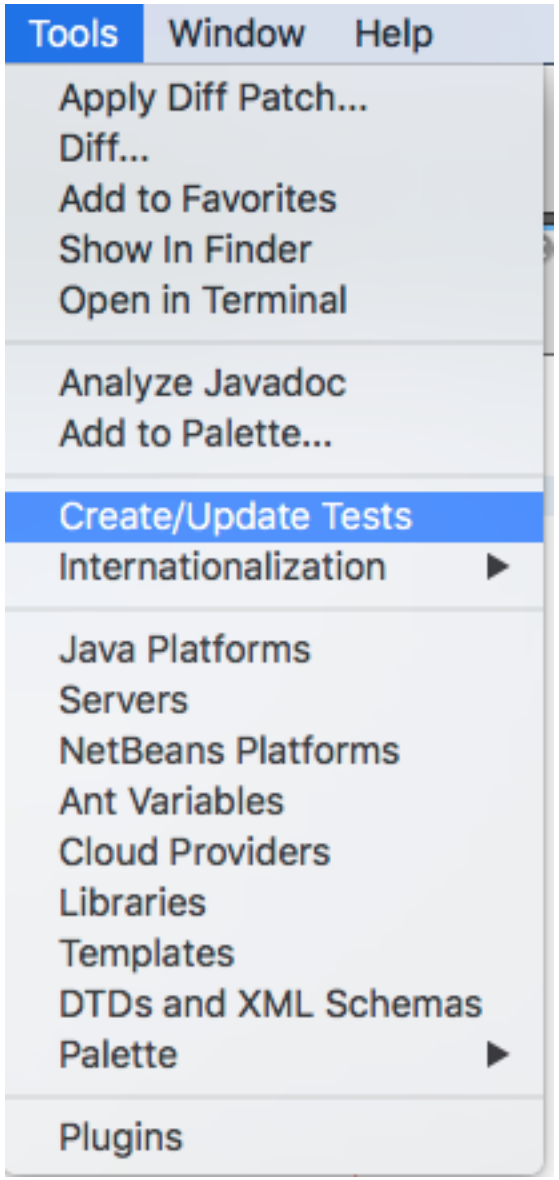
Java Unit, **JUnit**, is a library for **unit tests**

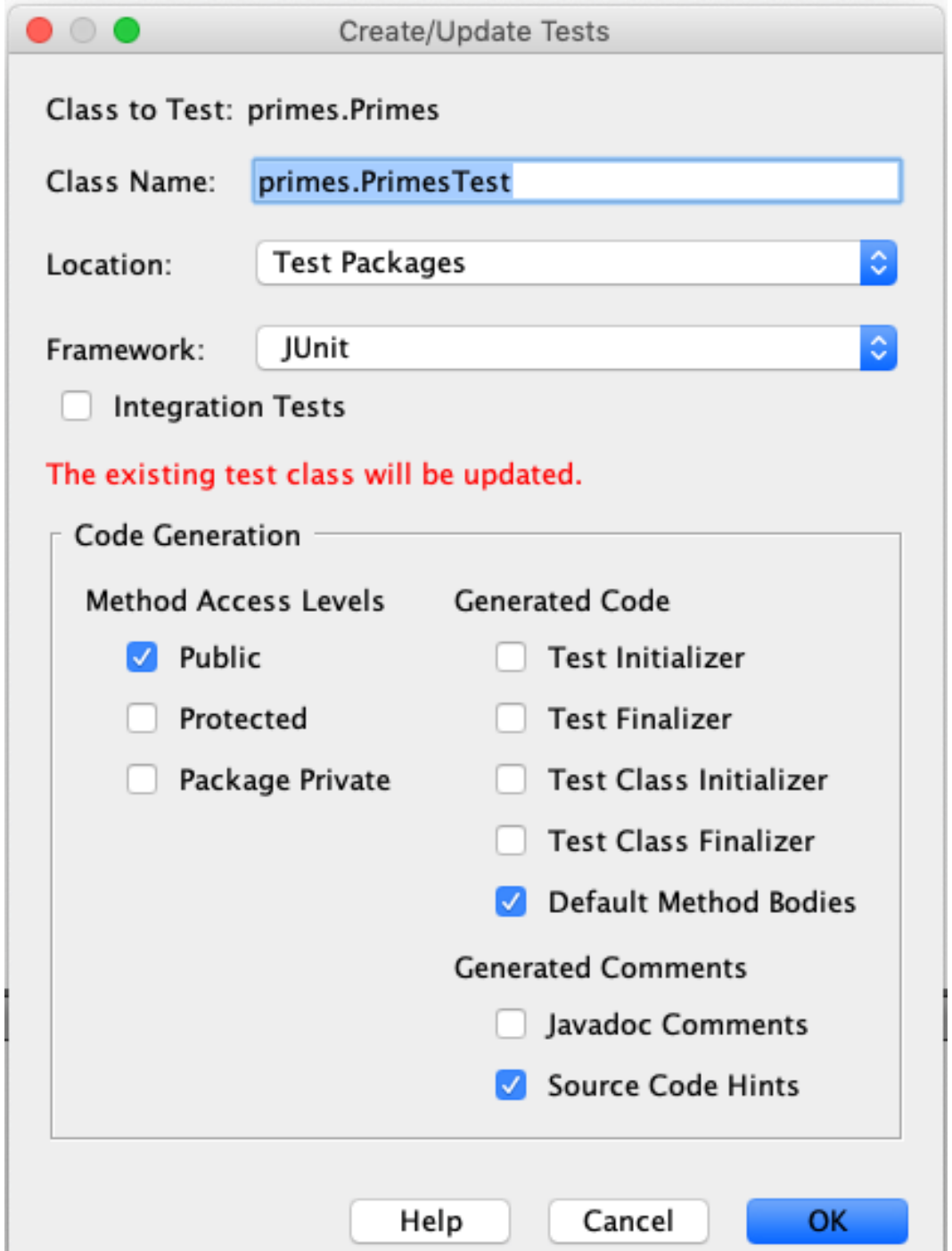
- JUnit is integrated in NetBeans (and all other major IDEs)
- JUnit test package is independent of the ordinary package
- packages can be executed independently

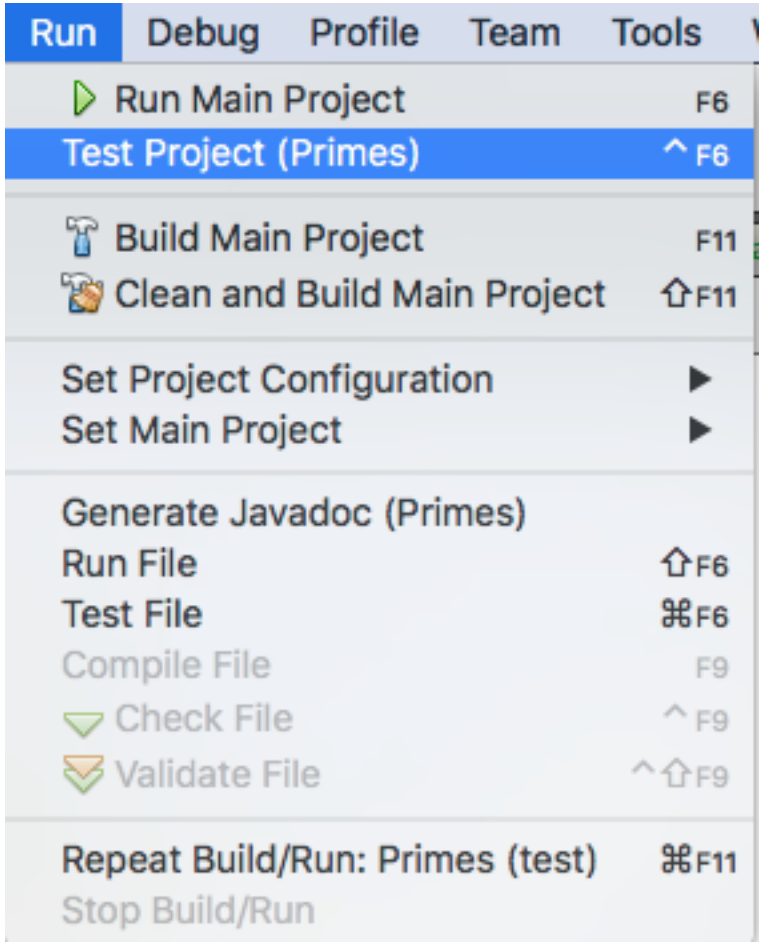
JUnit architecture



in NetBeans


Tools
Window
Help
Apply Diff Patch...
Diff...
Add to Favorites
Show In Finder
Open in Terminal
Analyze Javadoc
Add to Palette...
Create/Update Tests
Internationalization
Java Platforms
Servers
NetBeans Platforms
Ant Variables
Cloud Providers
Libraries
Templates
DTDs and XML Schemas
Palette
Plugins

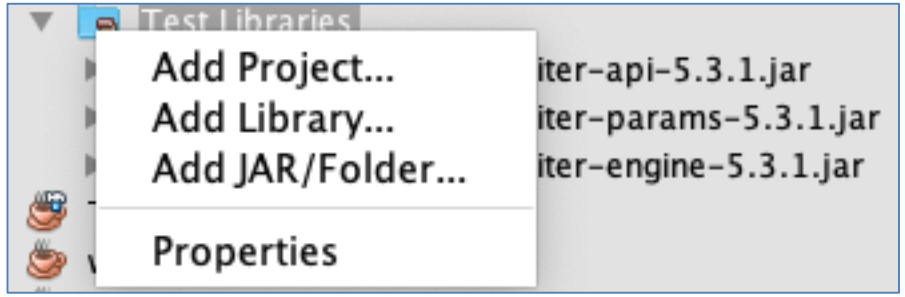

Create/Update Tests
Class to Test: primes.Primes
Class Name: primes.PrimesTest
Location: Test Packages
Framework: JUnit
Integration Tests
The existing test class will be updated.
Code Generation
Method Access Levels
Public
Protected
Package Private
Generated Code
Test Initializer
Test Finalizer
Test Class Initializer
Test Class Finalizer
Default Method Bodies
Generated Comments
Javadoc Comments
Source Code Hints
Help
Cancel
OK


Run
Debug
Profile
Team
Tools
Run Main Project F6
Test Project (Primes) ^F6
Build Main Project F11
Clean and Build Main Project ⇧F11
Set Project Configuration
Set Main Project
Generate Javadoc (Primes)
Run File ⇧F6
Test File ⌘F6
Compile File F9
Check File ^F9
Validate File ^⇧F9
Repeat Build/Run: Primes (test) ⌘F11
Stop Build/Run

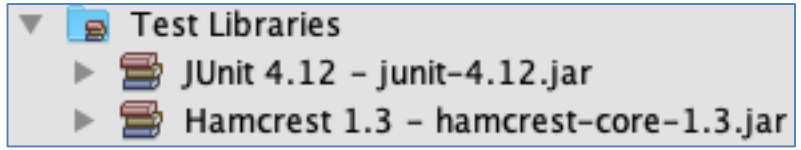
nasty details

Java is in the transition from JUnit 4 to JUnit 5
JUnit 5 is assumed to be available in Netbeans 11/12, but does not work properly
solution: use JUnit 4:

1. add `Test Libraries` (right click item in projects):
   `JUnit 4.12`
   `Hamcrest 1.3`
   they are part of the NetBeans download
   You can remove JUnit 5 (5.5.2)



2. replace imports in the generated test class by
```
import static org.junit.Assert.*;
import org.junit.*;
```

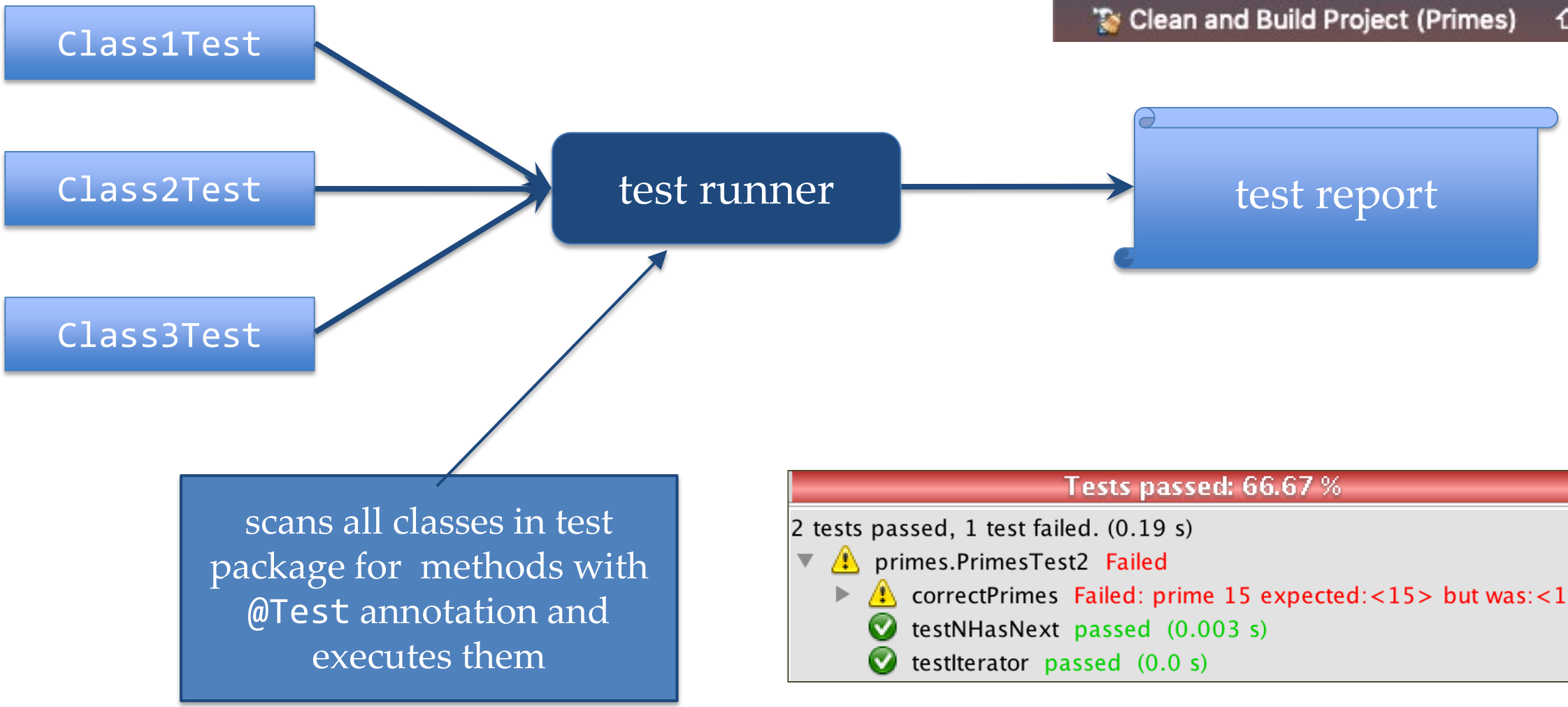


3. alternative to step 2: delete & regenerate test class

this should work without further issues.

test execution

test package



test runner (provided by JUnit)

execute each test method in the package. In pseudo code

```
for each test class TC in the test package {
   TC tc = new TC();
   tc.setUpClass();
   for each method @Test public void tm() in TC {
      tc.setUp();
      tc.tm();
      tc.tearDown();
   }
   tc.tearDownClass();
}
```

actually there are different test runners, we will only use the standard one

e.g. open database

the actual tests

for the default test runner all asserts in one method count as one test, it stops testing this method after the first fail

all `setUp` and `tearDown` methods may be absent

bread and butter of JUnit tests: asserts

useful methods of JUnit:

```
assertEquals  (                Object x, Object y )
assertEquals  ( String info, Object x, Object y )
assertEquals  ( long x, long y )
assertEquals  ( double x, double y, double delta )
assertTrue    ( boolean c )
assertFalse   ( boolean c )
assertSame    ( Object o, Object p )
assertNotSame ( Object o, Object p )
assertNull    ( Object o )
assertNotNull ( Object o )
fail          ( )
...
```

all methods with and without string

`pass()` is missing, it is implicit

full list at: http://junit.sourceforge.net/javadoc/org/junit/Assert.html

example class to test

```
public class PrimeNumberGenerator implements Iterable<Integer> {
  private final List<Integer> primeNumbers = new ArrayList<>(Arrays.asList(2,3));

   private boolean isPrime(int q) {
      for (int p : primeNumbers) {
         if (q % p == 0) {
            return false;
         }
      }
      return true;
   }

   @Override
   public Iterator<Integer> iterator() {
      return new Iterator<Integer>() {
         @Override
         public boolean hasNext() { … }

         @Override
         public Integer next() { … }
      };
   }
}
```

we can only test this method

what is generated for class `PrimeNumberGenerator` with method `iterator`?

```
public class PrimeNumberGeneratorTest {
  public PrimeNumberGeneratorTest() { }
  @Before
  public void setUp() { }
  @Test
  public void testIterator() {
    System.out.println("iterator");
    PrimeNumberGenerator instance = new PrimeNumberGenerator();
    Iterator<Integer> expResult = null;
    Iterator<Integer> result = instance.iterator();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
  }
}
```

Java annotations

an annotation is syntactic metadata that can be added to classes, methods, variables, parameters and packages

we have seen the `@Override` annotation

there are also annotations used by tools, like JUnit
`@Test @Before @After @BeforeClass @AfterClass ..`

@Before and @After methods

as many @Before and @After methods as you want

warning: execution order is unknown !

you can inherit @Before and @After methods from a superclass

- execution order is as follows:
  1. the @Before methods of the superclass
  2. the @Before methods of this class
  3. the @Test method of this class
  4. the @After methods of this class
  5. the @After methods of the superclass

WRITING UNIT TESTS

first tailored test class

```
public class PrimeNumberGeneratorTest {
  private PrimeNumberGenerator primes;
  private Iterator<Integer> it;
  public PrimeNumberGeneratorTest() {
     primes = new PrimeNumberGenerator();
  }

  @Before
  public void setUp() { it = primes.iterator(); }

  @Test
  public void testIterator() {
    System.out.println("iterator");
    assertNotNull("iterator", it);
  }

  @Test
  public void testIteratorHasNext() {
    System.out.println("iterator.hasNext");
    assertTrue("iterator.hasNext", it.hasNext());
  }
}
```

attributes used in this class

constructor of test class

gives each test a fresh iterator

do we have an iterator ?

does the iterator work as expected, i.e. does it have a number ?

optional description of assertion

useful tests

do we have an iterator?

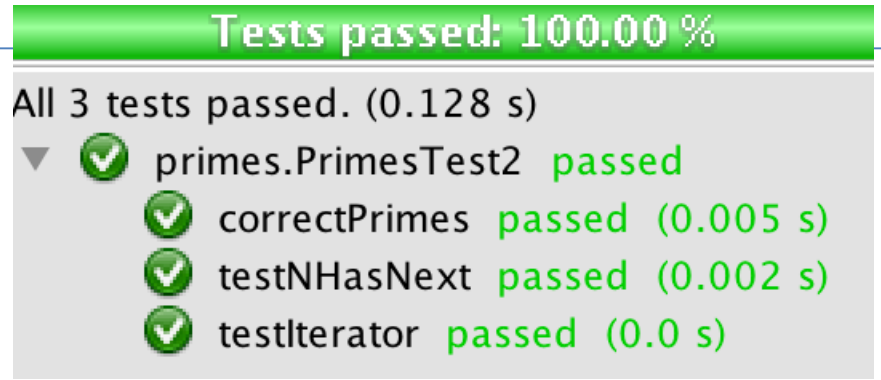
```
@Test
public void testIterator() {
  System.out.println("iterator");
  assertNotNull("iterator", it);
}
```

can the iterator produce N elements?

```
@Test
public void testNHasNext() {
  System.out.println("testNHasNext");
  final int N = 100;
  for (int i = 0; i < N; i += 1) {
    assertTrue("testNhasNext", it.hasNext());
    it.next();
  }
}
```

are the first few primes correct?

```
@Test
public void correctPrimes() {
  int [] knownPrimes = {2,3,5,7,11,13,15};
  for (Integer p : knownPrimes) {
    assertEquals("prime " + p, p, it.next());
  }
}
```



what should we test?

today we focus on functional tests:
does the software what it is supposed to do?

- how do we know what the software is supposed to do?
- we often has a partial specification by input/output examples, use these examples as test cases

other aspects that can be tested:

- performance,
- maintainability,
- stress-test,
- robustness, etc.

what should we test?

development of smart tests requires a course on its own

- rumour: Microsoft employs more testers than developers

test every method with complex behaviour

- no simple getters, simple setters, simple constructors, methods that call a single other method

some guidelines for test case development:

- **normal behaviour**
- boundary values of normal behaviour
- strange combinations of inputs
- where the documentation says "*never do ..*"
- wrong inputs (if we know what to expect)

recap: basic JUnit

JUnit executes all methods annotated by **`@Test`** in your test package

the test library counts ***fails***, ***passes*** and ***errors***

executing `fail()` or `fail ("message")` increases the number of fails

most used operation: `assertEquals( Object expected, Object actual )`

this basically executes

```
if ( expected.equals( actual ))
      pass();
else
      fail();
```

TYPE CASTING & ASSERTEQUALS

type casting

change of type of an object reference

- try to avoid this, but sometimes you really need it

up-casting:

- conversion from a subtype to a super-type (closer to `Object`)

```
Person p = (Person) new Student("Alice", 7);
```

- always allowed in Java, mostly left implicit

down-casting

- conversion from a super-type to a subtype

```
Person p = new Student("Alice",7);
..
Student s = (Student) p;
```

- must always be explicit
- fails (throw a `ClassCastException`) when casting to a type that is not on the path from `Object` to the actual type (e.g. (Integer) a.toString())

type cast is not type conversion

```
@Test
public void test5() {
  Integer five = 5;
  assertFalse("five.equals(\"5\")", five.equals("5"));
}
```

this is not a type error !

```
@Test
public void test5a() {
  Integer five = 5;
  assertEquals("five equals 5", five, "5");
}
```

this is not a type error !

confusing message

the `equals` method from Object

the Object class contains

```
public boolean equals(Object o) {
  return this == o;
}
```

this compares *object references*, this is usually not what we want

we want to compare the ***contents*** (attribute values)
of the given object and `this` object!

1. check if the argument `o` ≠ **`null`**
2. check if the `Object o` has the right type
3. ***down cast*** it to the actual type
4. compare arguments of the objects

example equals

```
public class Circle {
  private int x, y, r;

  public Circle(int x, int y, int r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }

  @Override
  public boolean equals(Object o) {
    if (o != null && getClass() == o.getClass()) {
      Circle c = (Circle) o;
      return x == c.x && y == c.y && r == c.r;
    } else {
      return false;
    }
  }
}
```

1: is there an object?

2: is object my type?

3: cast object to Circle

4: compare circle attributes

basic JUnit tests

```
public class CircleTest {
  private Circle c1, c2;
  @Before
  public void setUp() {
    c1 = new Circle(1, 2, 3);
    c2 = new Circle(1, 2, 3);
  }
  @Test
  public void testEq() {
    assertFalse("c1 == c2", c1 == c2);
  }
  @Test
  public void testEquals() {
    assertEquals("equals", c1, c2);
  }
}
```

executed for each test

every method labelled `@Test` is executed by JUnit

optional label, use it !

adding a subclass

```
public class ColoredCircle extends Circle {
  private Color color;

  public ColoredCircle(int x, int y, int r, Color c) {
    super(x, y, r);
    color = c;
  }
}
public class ColoredCircleTest {
  @Test
  public void testEquals() {
    Circle c1  = new Circle(1, 2, 3);
    Circle cc2 = new ColoredCircle(1, 2, 3, Color.red);
    assertEquals("equals", c1, cc2);
  }
}
```

this test fails

making `equals` succeed for a subtype

```
@Override
public boolean equals(Object o) {
  if (o != null && getClass() == o.getClass()) {
    Circle c = (Circle) o;
    return x == c.x && y == c.y && r == c.r;
  } else {
    return false;
  }
}
```

instanceof

`assertTrue` or `assertEquals`?

we can define

```
assertEquals("equals", c1, c2);
```

or

```
assertTrue("true", c1.equals(c2));
```

any difference?

effect is identical,
only the message for a failing test is better in the `assertEquals`

SOME MORE TESTING: MATHS

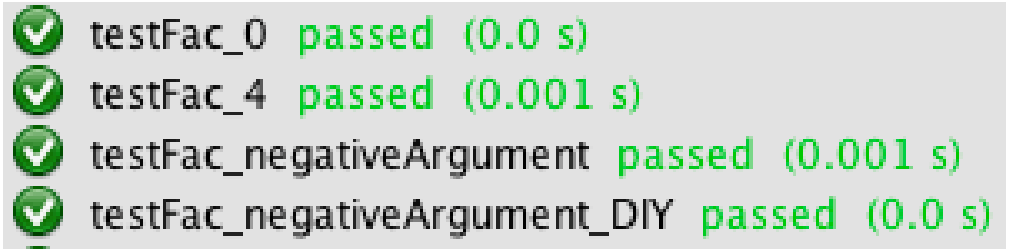
a new class to test with factorial method

```
public class MyMath {

  public static int fac (int a) {
    if (a < 0) {
      throw new IllegalArgumentException(
                        "fac(n): n should be >= 0, but was " + n);
    } else {
      int r = 1;
      for (int n = a ; n > 0; n -= 1)
        r *= n;
      return r;
    }
  }
  ..
```

tests for `fac` method

```
public class MyMathTest {
  @Test
  public void testFac_0() {
    int n = 0;
    int expResult = 1;
    int result = MyMath.fac( n );
    assertEquals("fac(0)", expResult, result);
  }
  @Test
  public void testFac_4() {
    assertEquals("fac(4)", 24, MyMath.fac( 4 ));
  }
  @Test
  public void testFac_negativeArgument_DIY() {
    try {
      MyMath.fac( -1 );
      fail("fac( -1 ): exception expected.");
    } catch (IllegalArgumentException e) {
      // implicit pass();
    }
  }
  @Test (expected = IllegalArgumentException.class)
  public void testFac_negativeArgument() {
    MyMath.fac( -1 );
  }
```



fairly complete set of tests:
- border case; 0
- normal case; 4
- wrong argument; -1

the better way to test exceptions

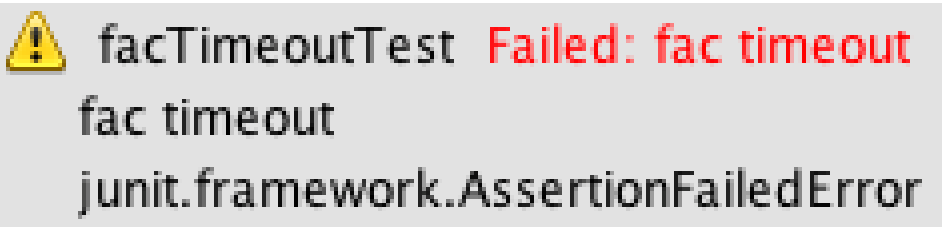
fails without exception

timeout: some computations take too long

catch them by an argument of @Test

max duration in ms

```
@Test (timeout = 10)
public void facTimeoutTest () {
  assertTrue("fac timeout", MyMath.fac(1000) > 1);
}
```

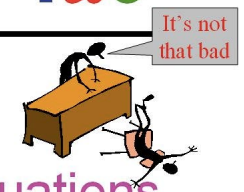


another method: solutions of a quadratic formula

```
public static List<Double> solutions(double a, double b, double c) {
  List<Double> list = new ArrayList<>(2);
  if (0 != a) {
    double d = b * b - 4.0 * a * c;
    if (0 == d) {
      list.add(-b / (2 * a));
    } else if (0 < d) { // do not forget this test
      list.add((-b - Math.sqrt(d)) / (2 * a));
      list.add((-b + Math.sqrt(d)) / (2 * a));
    }
  } else if (0 != b) {
    list.add(-c / b);
  }
  return list;
}
```

The Quadratic Formula ...

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

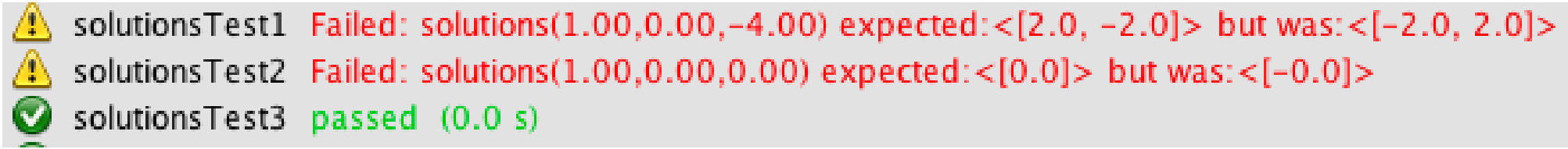


For Quadratic Equations

$ax^2 + bx + c = 0$

tests for `solutions`

```
@Test
public void solutionsTest1() {
  double a = 1, b = 0, c = -4;
  Double[] sol = {2.0, -2.0};
  String label = String.format("solutions(%f,%f,%f)", a, b, c);
  assertEquals(label, Arrays.asList(sol), MyMath.solutions(a, b, c));
}
@Test
public void solutionsTest2() {
  double a = 1, b = 0, c = 0;
  Double[] sol = {0.0};
  String label = String.format("solutions(%f,%f,%f)", a, b, c);
  assertEquals(label, Arrays.asList(sol), MyMath.solutions(a, b, c));
}
@Test
public void solutionsTest3() {
  double a = 0, b = 0, c = 0;
  Double[] sol = {};
  String label = String.format("solutions(%f,%f,%f)", a, b, c);
  assertEquals(label, Arrays.asList(sol), MyMath.solutions(a, b, c));
}
```



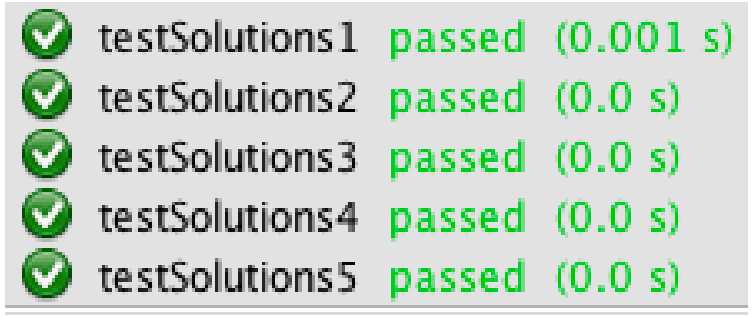
better tests correctness of `solutions`

```
private void testSolutions(double a, double b, double c) {
  String s = String.format("solutions(%f,%f,%f) ", a, b, c);
  double delta = 0.0001; // allowed error in double computation
  List<Double> l = MyMath.solutions(a, b, c);
  if (l.isEmpty()) {
    assertTrue(s + "no solution",
     (0 != a && 4 * a * c > b * b) || (0 == a && 0 == b));
  } else {
    for (Double x : l) {
      assertEquals(s + "solution " + x, 0.0, a * x * x + b * x + c, delta);
      assertEquals(s + "unique " + x, 1, Collections.frequency(l, x));
    }
  }
}
```

is this a complete correctness test?

better tests for `solutions`: actual tests

```
@Test
public void testSolutions1() {
  testSolutions(0, 0, 0);
}
@Test
public void testSolutions2() {
  testSolutions(1, 0, -4);
}
@Test
public void testSolutions3() {
  testSolutions(1, -4, 4);
}
@Test
public void testSolutions4() {
  testSolutions(1, 2, 3);
}
@Test
public void testSolutions5() {
  testSolutions(0, 2, 3);
}
```



better tests for `solutions`: actual tests

is it better to write the test below?

```
@Test
public void testSolutions() {
  testSolutions(0, 0, 0);
  testSolutions(1, 0, -4);
  testSolutions(1, -4, 4);
  testSolutions(1, 2, 3);
  testSolutions(0, 2, 3);
}
```

more compact,
but JUnit stops at the first failure in a test method

potentially fewer tests are executed; fix one bug, only then find the next

TESTING INTERNAL CLASS STATE

behaviour vs. state testing

the JUnit class is outside the tested class:
tests have no access to private attributes of tested objects

- attributes of classes should be private

hence, we can only test the effect of the state change in the behaviour instead of the state change itself

example: `AtSchool` class with state

```
public class AtSchool {
  private List<String> bag = new ArrayList<>();

  public AtSchool add(String item) {
    if (null != item && !item.isEmpty() && !bag.contains(item)) {
      bag.add(item);
    }
    return this;
  }
  public boolean remove(String item) {
    return bag.remove(item);
  }
  public int count() {
    return bag.size();
  }
  public String[] getItems() {
    bag.sort((a,b) -> a.compareTo(b));
    return bag.toArray(new String[0]);
  }
  @Override
  public String toString() {
    return "AtSchool " + bag;
  }
}
```

state

state change

state change

state access

Atschool JUnit tests: setup + add

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class AtSchoolTest {
  private AtSchool s0, s1, s2;
  private final String laptop = "Laptop", book = "Book", pen = "Pen";

  @Before
  public void setUp() {
    s0 = new AtSchool();
    s1 = new AtSchool().add(book);
    s2 = new AtSchool().add(pen).add(book);
  }

  @Test
  public void testAdd_Laptop_s0() {
    s0.add(laptop);
    List<String> items = Arrays.asList(s0.getItems());
    assertTrue("s0 add "+ laptop, items.contains(laptop));
  }

  @Test
  public void testAdd_Laptop_s1() {
    s1.add(laptop);
    List<String> items = Arrays.asList(s1.getItems());
    assertTrue("s1 add "+ laptop,items.contains(laptop));
  }
```

Atschool JUnit tests: add special cases

```
@Test
public void testAdd_Empty_s1() {
  String item = "";
  s1.add(item);
  assertEquals("s1 add "+ item, 1, s1.count());
}

@Test
public void testAdd_null_s1() {
  String item = null;
  s1.add(item);
  assertEquals("s1 add " + item, 1, s1.co
}

@Test
public void testAdd_Book_s1() {
  String item = "Book";
  s1.add(item);
  assertEquals("s1 add "+ book, 1, s1.count());
}
```

good method names and assertion info are better than System.out.println

Atschool JUnit tests: remove

```
@Test
public void testRemove_s0_Book() {
  assertFalse("s0 remove " + book, s0.remove(book));
  assertEquals("s0 remove " + book, 0, s0.count());
}

@Test
public void testRemove_s1_Book() {
  assertTrue("s1 remove " + book, s1.remove(book));
  assertEquals("s1 removed " + book, 0, s1.count());
}

@Test
public void testRemove_s2_Book() {
  assertTrue("s2 remove " + book, s2.remove(book));
  assertEquals("s2 remove " + book, 1, s2.count());
}

@Test
public void testRemove_s2_Laptop() {
  assertFalse("s2 remove " + laptop, s2.remove(laptop));
  assertEquals("s2 removed " + laptop, 2, s2.count());
}
```

test both result and effect

Atschool JUnit tests: count + toString

```
@Test
public void testCount_s2() {
  assertEquals("s2.count", 2, s2.count());
}

@Test
public void testToString_s2_getItems() {
  String s = s2.toString();
  for (String i: s2.getItems()) {
    assertTrue("s2.toString contains " + i, s.contains(i));
  }
}
@Test
public void testToString_s2() {
  String[] items = {book, pen};
  String s = s2.toString();
  for (String i: items) {
    assertTrue("s2.toString contains " + i, s.contains(i));
  }
}
```

TEST RESULTS

verdicts in JUnit

a verdict is the result of executing a single test

- Pass: the test case is executed, and the software under test performed as expected
- Fail: the test case is executed, but the software did ***not*** perform as expected
  - typically the `equals` in `assertEquals` yields `false`
- Error: the test case is not executed as intended
  - potential reasons:
    1. an exception is thrown in the test case;
    2. the test took too long (timeout)
    3. a runtime error occurred (also causes an exception);
    4. the test case could not be set up properly

JUnit reports a fail, what next?

1. THINK!
   - try to come up with an explanation of the observed results
   - is it a problem with the test, or the program?
2. if this does not reveal the problem:
   - write additional test to pinpoint the problem
     - e.g. check intermediate steps
     - these tests are useful in the future
3. explain to another person why your set of tests is complete, what works correctly and what fails
   - (no person required: a rubber duck also works)
4. use the debugger to observe internal states

there is more …

- assertions
- matchers and `assertThat`
- parametric tests
- JUnit-QuickCheck
- JUnit 5
- test automation in e.g. Github, Gitlab, etc.
- …

effect of (unit) testing

a SUT that passes all test is not necessarily correct

- the tests are rarely exhaustive

failures in the tests does not necessarily imply that the SUT is incorrect

- also the tests can be incorrect
  - errors in the test, or
  - erroneous interpretation of specs

nevertheless testing is extremely useful

- formulating tests increases the understanding of the code
- testing reveals problems or increases the confidence
- JUnit documents the tests and facilitates regression tests

Lecture 8: GUIs: JavaFX (I)