Design Patterns

- Automated Testing
- JUnit
- Test-Driven Development
- Test Coverage
- Integration Tests

JUnit

Most popular automated test framework for Java.

Relatively easy to use.

Most other test frameworks draw heavily from it.

Something to Test

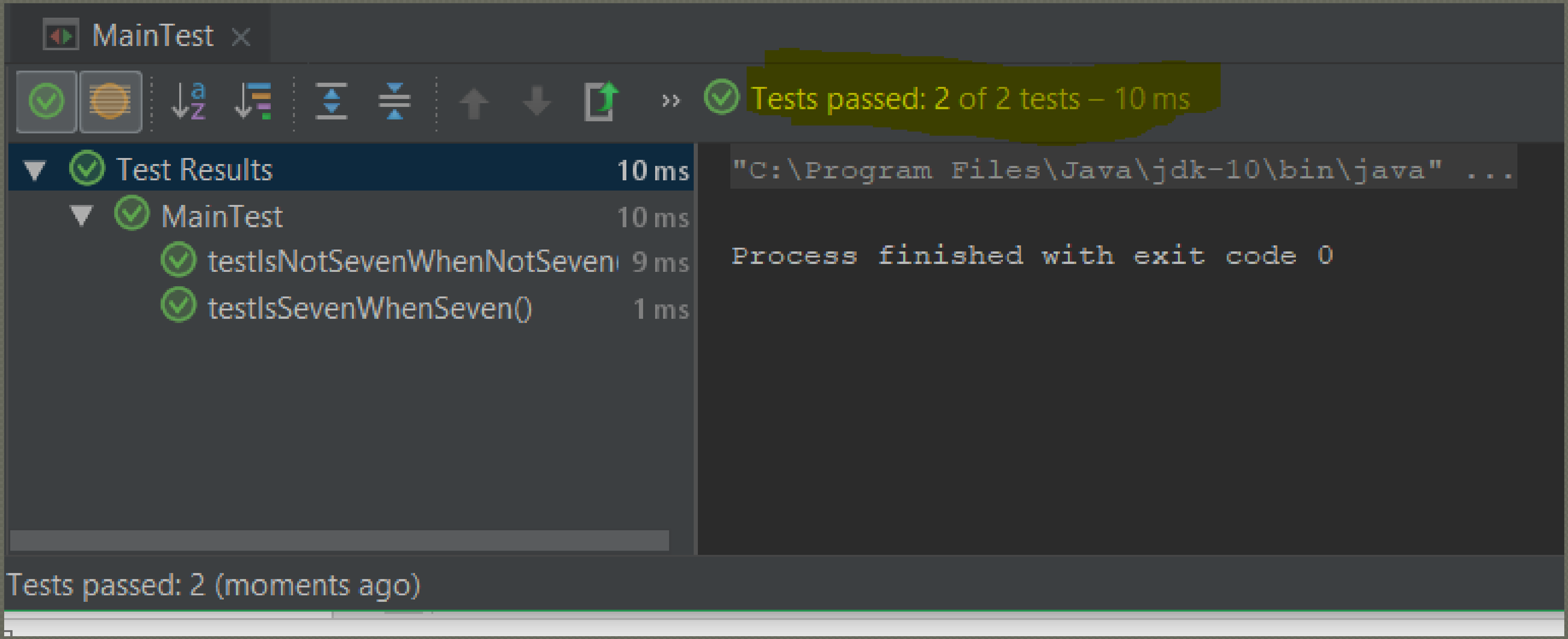
```
class Seven {
    static boolean isSeven( int i ) {
        return i == 7;
    }
}
```

The Tests

```
class MainTest {
    @Test
    void testIsSevenWhenSeven() {
        assertTrue( Seven.isSeven( 7 ) );
    }

    @Test
    void testIsNotSevenWhenNotSeven() {
        assertFalse( Seven.isSeven( 42 ) );
    }
}
```

Running the Tests



```
Tests passed: 2 of 2 tests – 10 ms

▼ Test Results                         10 ms
  ▼ MainTest                           10 ms
      testIsNotSevenWhenNotSeven        9 ms
      testIsSevenWhenSeven()            1 ms

"C:\Program Files\Java\jdk-10\bin\java" ...

Process finished with exit code 0

Tests passed: 2 (moments ago)
```

Asserts

- If any asserts in the test fail, the test fails
- `Assert()` takes a Boolean expression
  - It requires that the result be true
- `AssertEquals`, `AssertFalse`, etc. are simply asserts that produce nicer error messages
  - `Assert()` by itself is all you technically need
- Assert methods:
  - http://junit.sourceforge.net/javadoc/org/junit/Assert.htm
    l

How to Write a Test

- Every test should contain at least one assert, unless you are testing for an exception
  - uncaught exceptions fail tests too
- Every test is void and has no parameters
- Every test is marked with the attribute "@Test"
- You cannot assume that tests will run in a certain order
  - Keep this in mind for database testing
  - It's okay to set up a single instance to run a bunch of tests on, but don't assume they run in series!

The Game of Nim

```
class NimGame {
    int count = 0;
    int turn = 0;

    void increment( int by ) {
        //by can be at most 2:
        by = Math.max( by, 2 );

        //by must be at least 1:
        by = Math.min( by, 1 );

        count += by;
        turn++;
    }

    boolean gameOver() {
        return count >= 20;
    }
}
```

NimTest

```
class NimTest {
    @Test
    void testGameInitialize() {
        NimGame game = new NimGame();
        assertEquals( game.count, 0 );
        assertEquals( game.turn, 0 );
    }

    @Test
    void testIncrements() {
        NimGame game = new NimGame();
        game.increment(1 );
        assertEquals( game.count, 1 );
        game.increment(2);
        assertEquals( game.count, 3 );
        assertEquals( game.turn, 2 );
    }

    @Test
    void testIncrementsWithInvalidInput() {
        NimGame game = new NimGame();
        game.increment(-200 );
        assertEquals( game.count, 1 );
        game.increment(200000);
        assertEquals( game.count, 3 );
        assertEquals( game.turn, 2 );
    }
}
```

Guidelines

Use

Use the assertX methods instead of plain assert for nicer error messages.

- If you don't do this, you'll see that the test failed but not what values caused it to fail.

Don't cram

Don't cram too much into one test

- I'm pushing it in my example by making turn be tested along with increment.

Don't mix

Don't mix tests for different classes in unit tests

- This is a different kind of testing called "integration testing" which we'll talk about later.

Avoid

Avoid making tests depend on one another. Don't call tests from tests. Factor out common code into methods and call those.

A few things worth noting

- JUnit tests are just methods
  - They must follow Java rules
- You can’t (easily) assert exceptions
- Tests are for “your” code, not for libraries