哈尔滨工业大学（深圳）

面向对象的软件构造导论
实验指导书

实验三 JUnit 和单元测试

2022 春

目录

1. 实验目的

1. 了解单元测试的定义及其重要性,掌握使用 JUnit5 进行单元测试的方法;
2. 理解编码规范的重要性，熟悉代码规范插件的使用方法。

2. 实验环境

1. Windows 10
2. IntelliJ IDEA 2021.2.3
3. Java 11
4. JUnit5

3. 实验内容（2 学时）

（1）创建 JUnit5 单元测试类，使用 JUnit5 编写测试用例的常见用法，如注解、断言、假设、禁用测试等，结合飞机大战实例，为系统中类的设计测试用例，并编写单元测试代码;
（2）使用阿里编码规约插件扫描代码，并改正问题代码。

4. 实验步骤

单元测试（Unit Testing），是指对软件中的最小可测试单元进行检查和验证。对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

JUnit 5 是什么？

JUnit 是一个 Java 编程语言的单元测试框架。JUnit 在测试驱动的开发方面有很重要的发展，是起源于 JUnit 的一个统称为 xUnit 的单元测试框架之一。JUnit 促进了“先测试后编码”的理念，强调建立测试数据的一段代码，可以先测试，然后再应用。与以前的 JUnit 版本不同，JUnit 5 是由三个不同子项目的几个不同的模块组成。

JUnit 5 = JUnit Platform（基础平台） + JUnit Jupiter（朱庇特（主宰）、核心程序） + JUnit Vintage（老版本的支持）

JUnit Platform: 是在 JVM 上启动测试框架（launching testing frameworks）的基础。它还定义了用于开发平台上运行的测试框架的测试引擎（TestEngine）API。此外，该平台还提供了一个控制台启动器（Console Launcher），可以从命令行启动平台，并为 Gradle 和 Maven 构建插件，以及一个基于 JUnit 4 的运行器（JUnit 4 based Runner），用于在平台上运行任何 TestEngine 。

JUnit Jupiter: 是在 JUnit 5 中编写测试和扩展的新编程模型（ programming model ）和扩展模型（ extension model ）的组合。另外，Jupiter 子项目还提供了一个 TestEngine，用于在平台上运行基于 Jupiter 的测试。

JUnit Vintage: 提供一个在平台上运行 JUnit 3 和 JUnit 4 的 TestEngine 。

4.1 自动创建 JUnit5 单元测试类

(1) 在 IntelliJ IDEA 中创建一个 Java Project （本例中 project 名为 JunitDemo）,在 package junit.demo 下创建类 Calculator.java,在其中编写 5 个方法，用于将参数 x,y 的值进行加、减、乘、除并且返回结果。

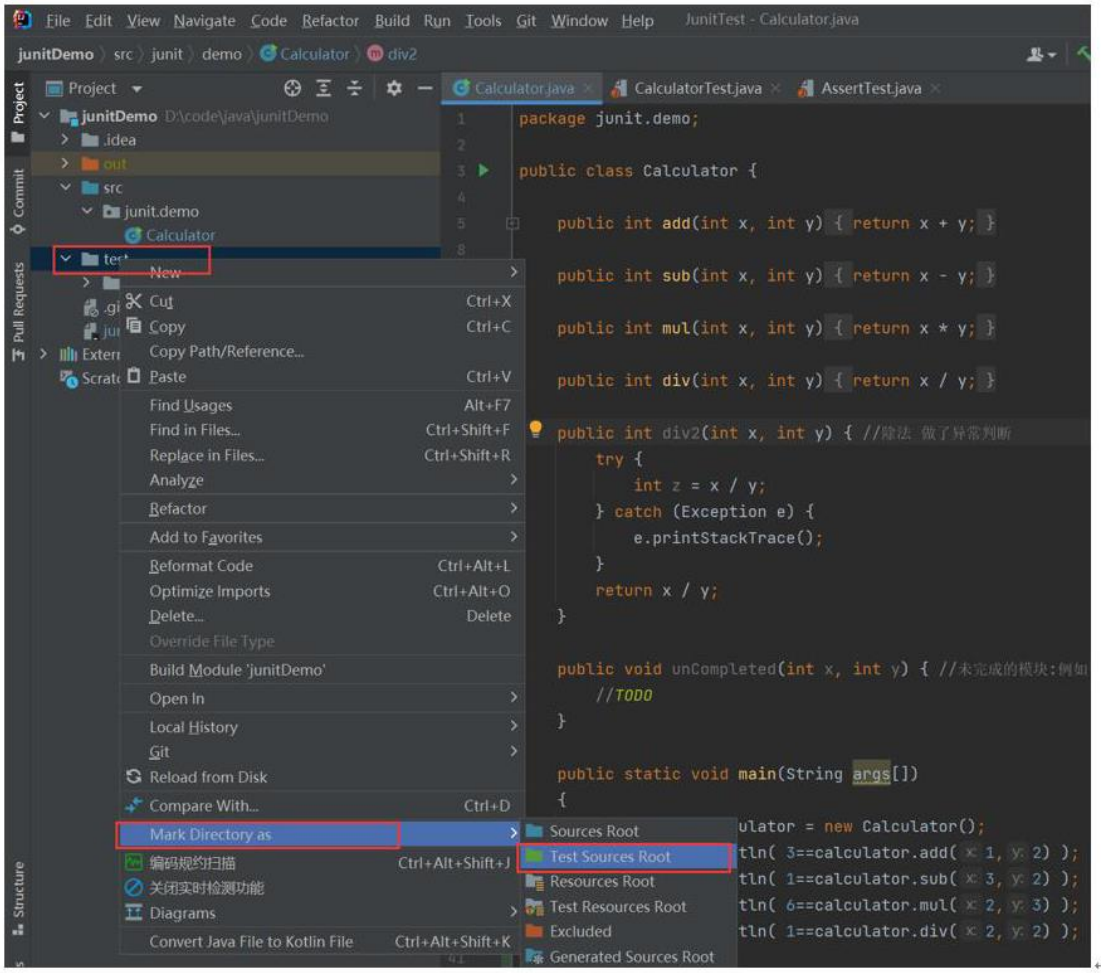
Calculator.java

```
package junit.demo;
public class Calculator {
    public int add(int x, int y) { //加法
        return x + y;
    }

    public int sub(int x, int y) { //减法
        return x - y;
    }

    public int mul(int x, int y) { //乘法
        return x * y;
    }

    public int div(int x, int y) { //除法
        return x / y;
    }

    public int div2(int x, int y) { //除法 做了异常判断
        try {
            int z = x / y;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return x / y;
    }

    public void unCompleted(int x, int y) { //未完成的模块:例如平方、开方等等
        //TODO
    }
```

```
}
```

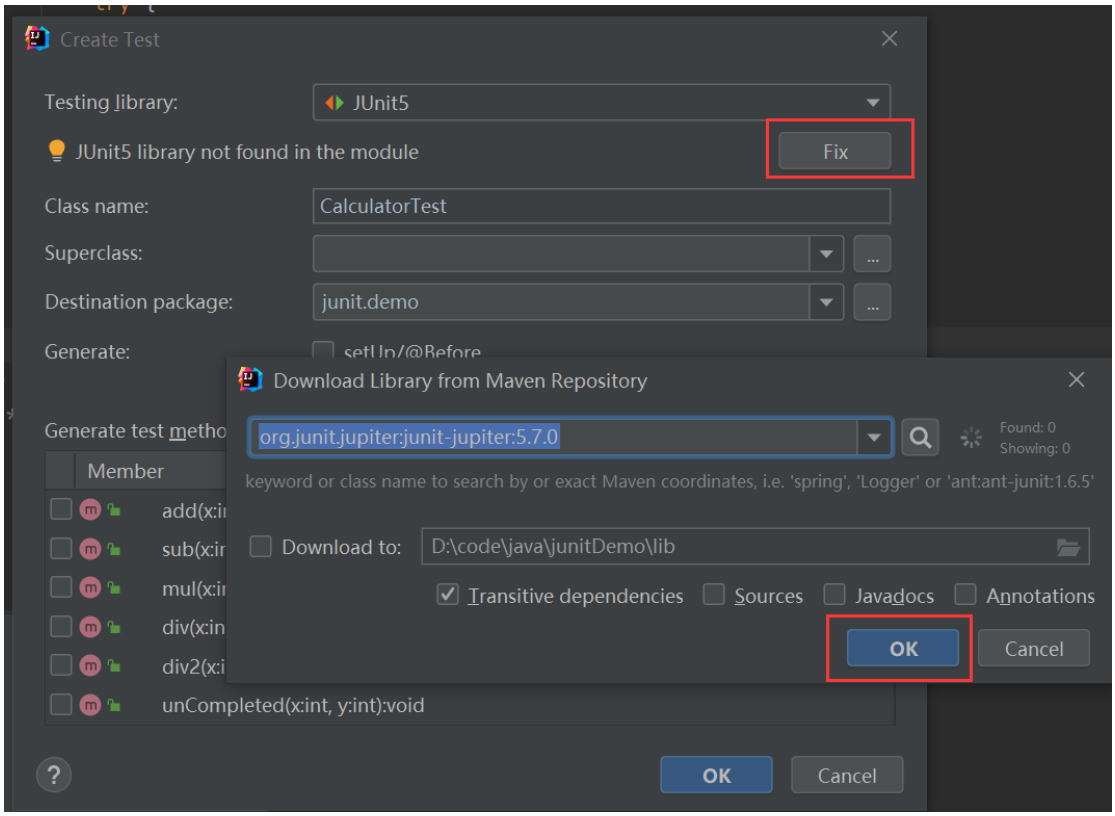
(2) 创建一个 test 的文件夹，用于存放测试函数，然后标记为测试类文件夹。

右键点击 test 文件夹，选择 Mark Directory as -> Test Sources Root

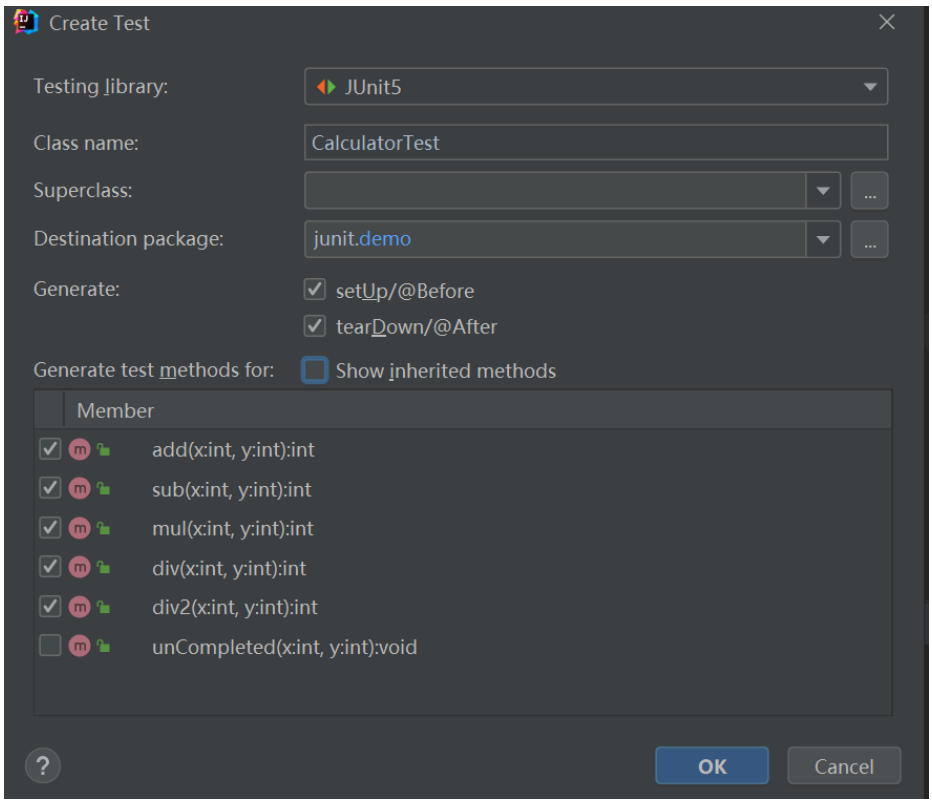


（3）打开 Calculator . java 文件，在类的内部同时按下快捷键 ctrl + shift + T，选择 Create New Test。

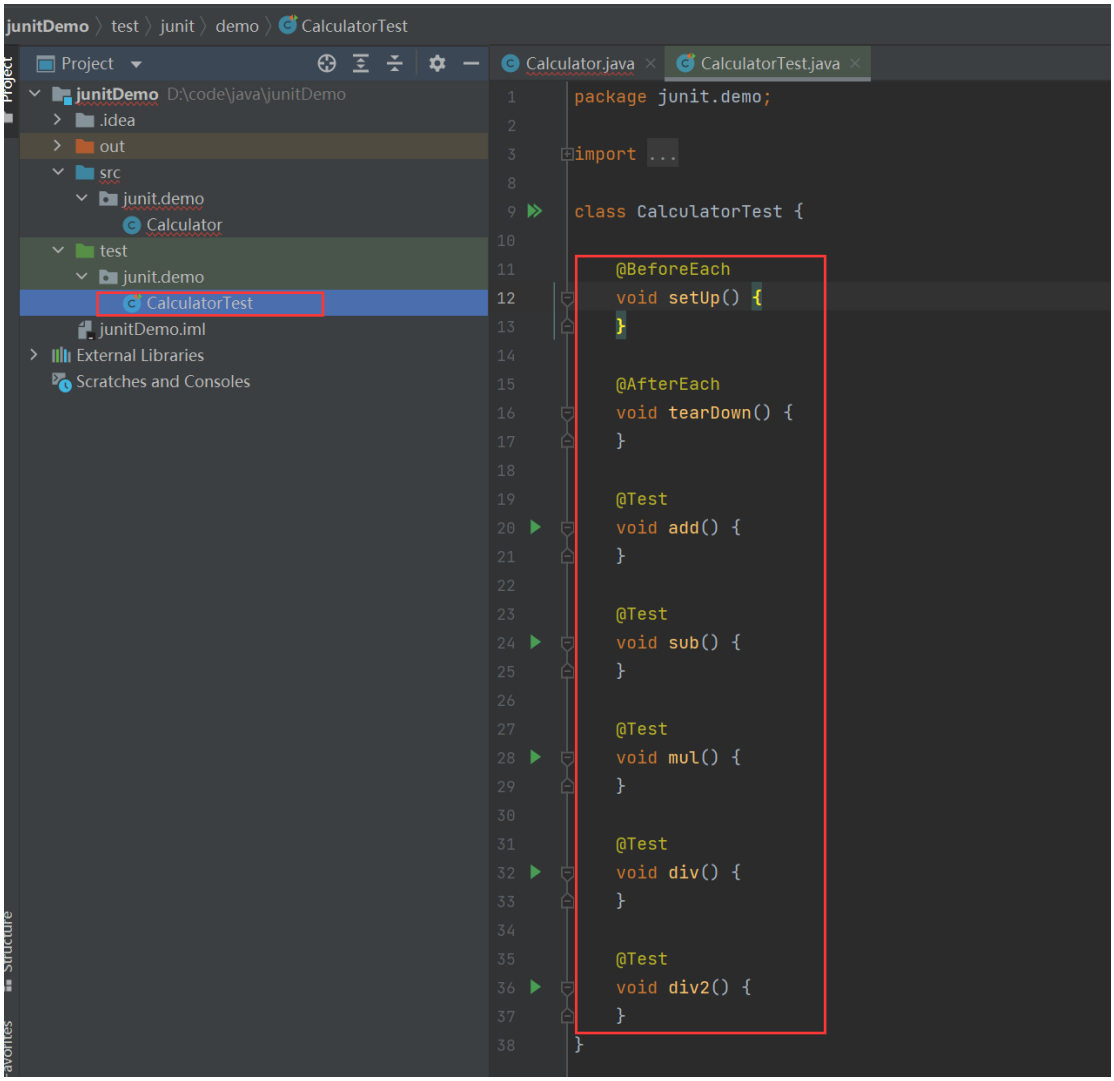
若之前没有安装过 JUnit5 Library， 在界面中点击 Fix 按钮，IDEA 会帮助你下载好 JUnit5（点击 OK）。



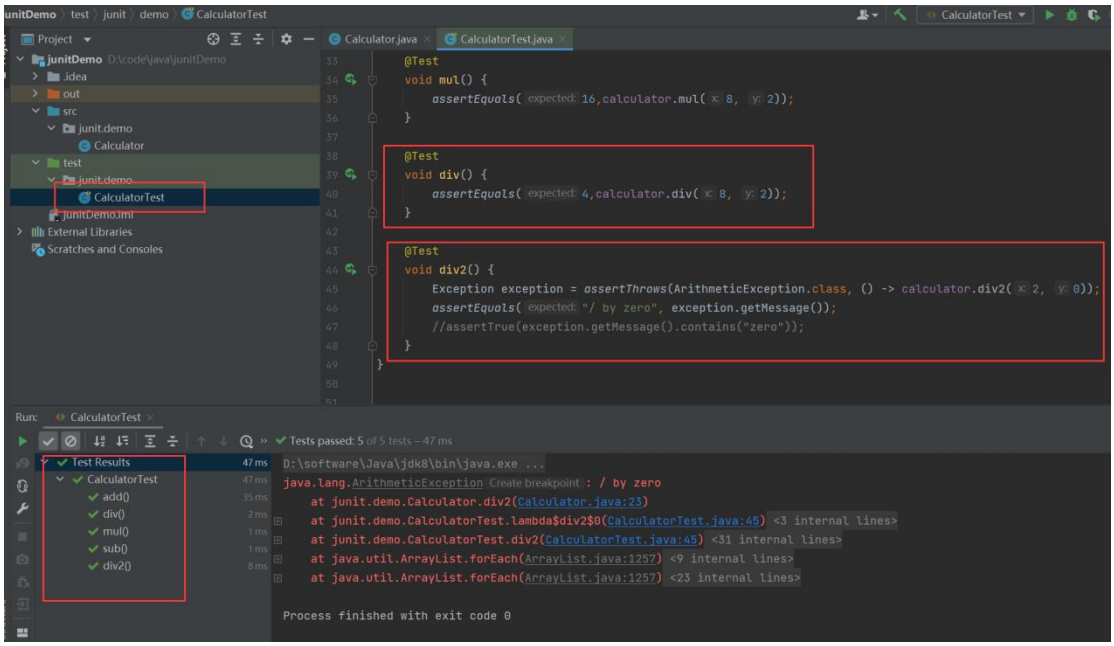
（4）安装后在 Member 中勾选 Calculator 中需要进行单元测试的方法。



（5）查看 test 目录下自动生成的单元测试类 CalculatorTest。



（6）修改单元测试代码后，右键 CalculatorTest 类，选择 Run CalculatorTest， 即可运行该单元测试类。



4.2 JUnit5 的常见用法

4.2.1 JUnit5 注解（Annotations)

下面列出了 JUnit5 提供的一些常用注解：

Annotation	Description
@Test	Denotes a test method
@DisplayName	Declares a custom display name for the test class or test method
@BeforeEach	Denotes that the annotated method should be executed before each test method
@AfterEach	Denotes that the annotated method should be executed after each test method
@BeforeAll	Denotes that the annotated method should be executed before all test methods
@AfterAll	Denotes that the annotated method should be executed after all test methods
@Disable	Used to disable a test class or test method
@Nested	Denotes that the annotated class is a nested, non-static test class
@Tag	Declare tags for filtering tests
@ExtendWith	Register custom extensions

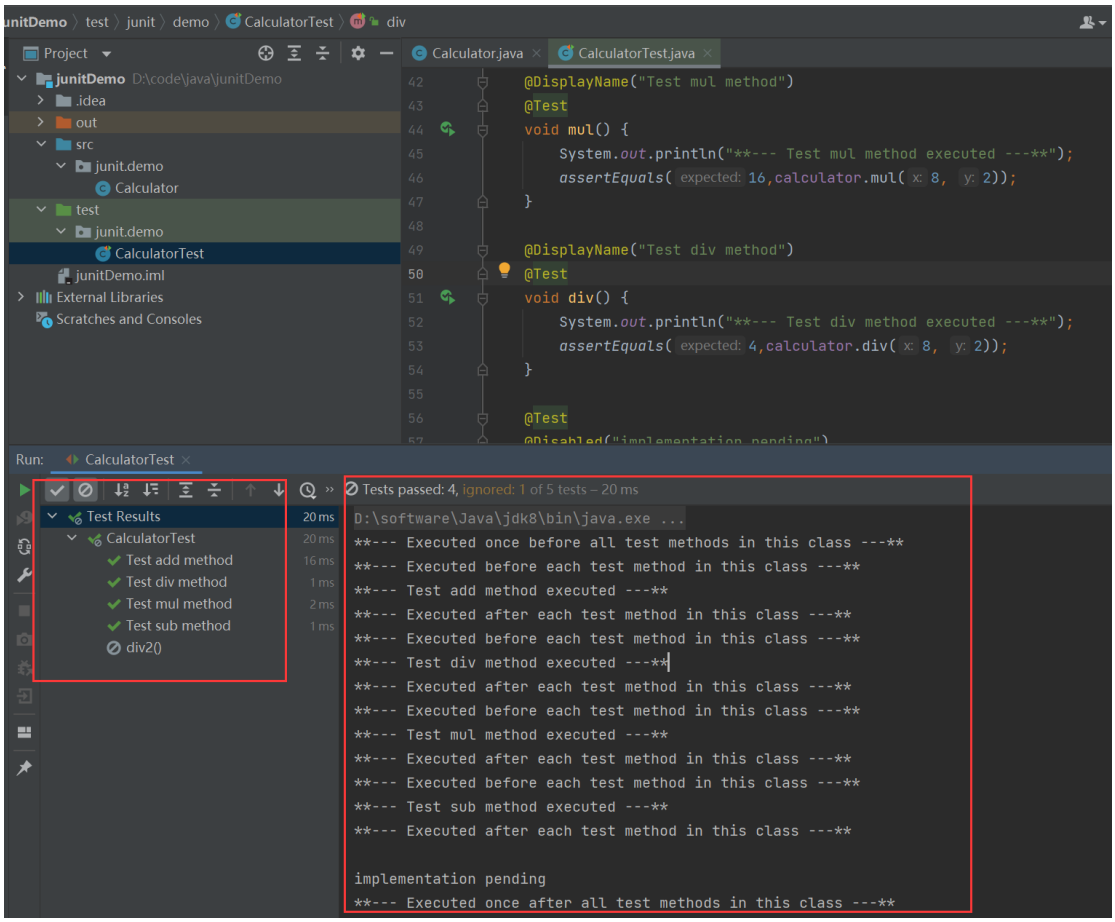
代码示例：

```
package junit.demo;

import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    private Calculator calculator;

    @BeforeAll
    static void beforeAll() {
        System.out.println("**--- Executed once before all test methods in this class ---**");
    }

    @BeforeEach
    void setUp() {
        System.out.println("**--- Executed before each test method in this class ---**");
        calculator = new Calculator();
    }

    @AfterEach
    void tearDown() {
        System.out.println("**--- Executed after each test method in this class ---**");
        calculator = null;
    }
```

```
    @DisplayName("Test add method")
    @Test
    void add() {
        System.out.println("**--- Test add method executed ---**");
        assertEquals(10,calculator.add(8, 2));
    }

    @DisplayName("Test sub method")
    @Test
    void sub() {
        System.out.println("**--- Test sub method executed ---**");
        assertEquals(6,calculator.sub(8, 2));
    }

    @DisplayName("Test mul method")
    @Test
    void mul() {
        System.out.println("**--- Test mul method executed ---**");
        assertEquals(16,calculator.mul(8, 2));
    }

    @DisplayName("Test div method")
    @Test
    void div() {
        System.out.println("**--- Test div method executed ---**");
        assertEquals(4,calculator.div(8, 2));
    }

    @Test
    @Disabled("implementation pending")
    void div2() {
    }

    @AfterAll
    static void afterAll() {
        System.out.println("**--- Executed once after all test methods in this class ---**");
    }

}
```

运行结果：

从测试结果中我们可以看到 div2 用例被禁用，没有被执行。



4.2.1 JUnit5 断言（Annotations）

必须使用断言将每个测试方法的条件评估为 true，以便测试可以继续执行。JUnit Jupiter 断言保存在 org.junit.jupiter.api.Assertions 类中，所有方法都是静态的。

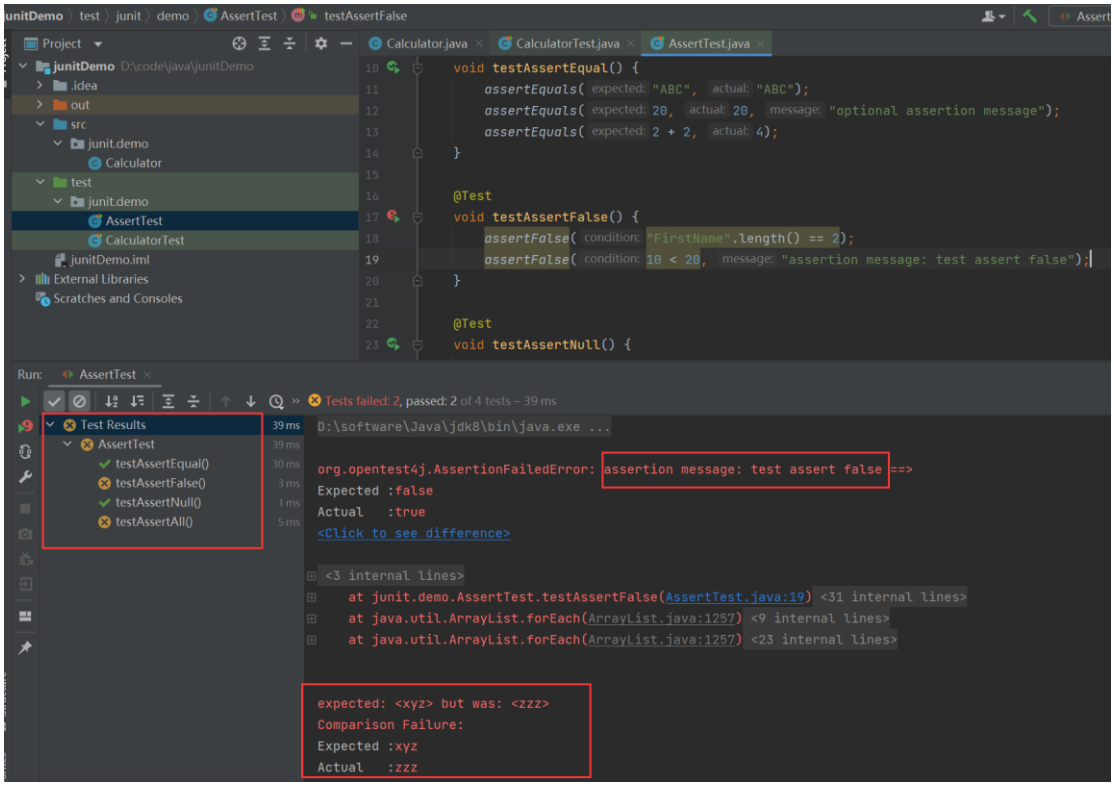
Assertion	Description
assertEquals(expected, actual)	Fails when expected does not equal actual
assertFalse(expression)	Fails when expression is not false
assertNull(actual)	Fails when actual is not null
assertNotNull(actual)	Fails when actual is null
assertAll()	Group many assertions and every assertion is executed even if one or more of them fails
assertTrue(expression)	Fails if expression is not true
assertThrows()	Class to be tested is expected to throw an exception

代码示例：

```
package junit.demo;
```

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class AssertTest {

	@Test
	void testAssertEqual() {
		assertEquals("ABC", "ABC");
		assertEquals(20, 20, "optional assertion message");
		assertEquals(2 + 2, 4);
	}

	@Test
	void testAssertFalse() {
		assertFalse("FirstName".length() == 2);
		assertFalse(10 < 20, "assertion message: test assert false");
	}

	@Test
	void testAssertNull() {
		String str1 = null;
		String str2 = "abc";
		assertNull(str1);
		assertNotNull(str2);
	}

	@Test
	void testAssertAll() {
		String str1 = "abc";
		String str2 = "pqr";
		String str3 = "xyz";
		assertAll("numbers",
				() -> assertEquals(str1,"abc"),
				() -> assertEquals(str2,"pqr"),
				() -> assertEquals(str3,"zzz")
		);
	}
}
```

运行结果：



4.2.3 JUnit5 假设（Assumptions)

假设是 org.junit.jupiter.api.Assumptions 类中的静态方法。他们仅在满足指定条件时执行测试，否则测试将中止。 中止的测试不会导致构建失败。 当假设失败时，将抛出 org.opentest4j.TestAbortedException 并跳过测试。

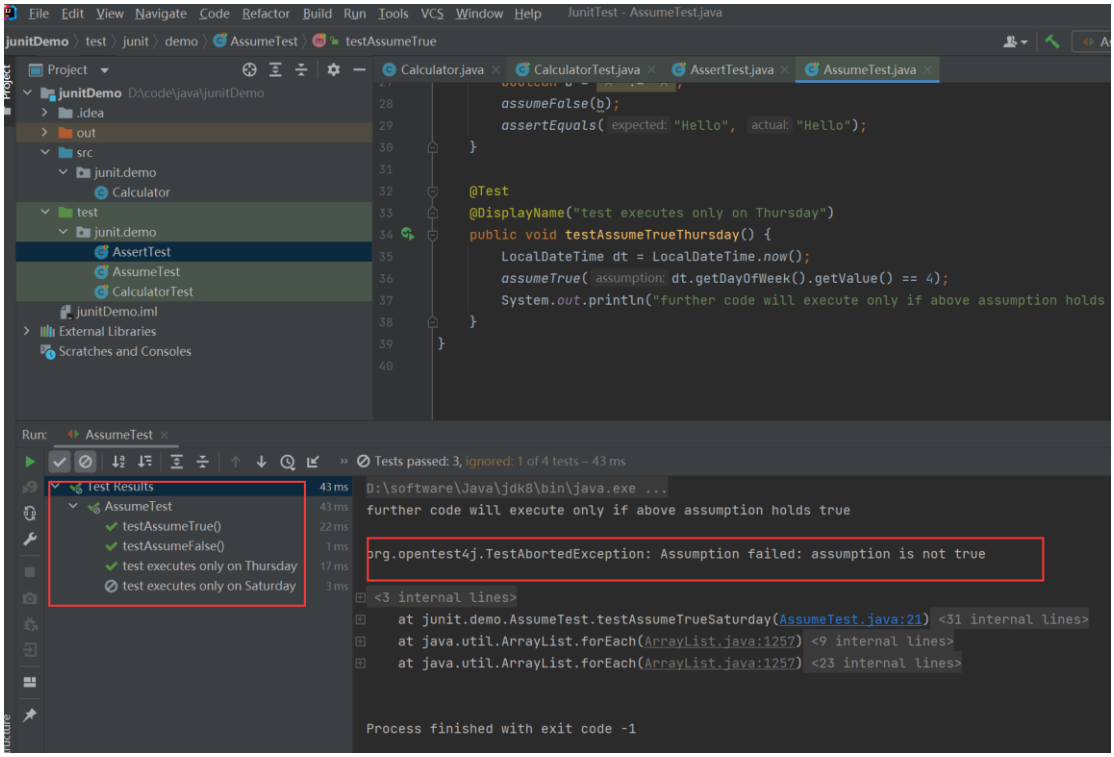
Assumptions	Description
assumeTrue	Execute the body of lamda when the positive condition hold else test will be skipped
assumeFalse	Execute the body of lamda when the negative condition hold else test will be skipped
assumingThat	Portion of the test method will execute if an assumption holds true and everything after the lambda will execute irrespective of the assumption in assumingThat() holds

代码示例：

```
package junit.demo;

import org.junit.jupiter.api.*;
import java.time.LocalDateTime;
import static org.junit.jupiter.api.Assertions.*;
import static    org.junit.jupiter.api.Assumptions.*;

public class AssumeTest {

    @Test
    void testAssumeTrue() {
        boolean b = 'A' == 'A';
```

```
        assumeTrue(b);
        assertEquals("Hello", "Hello");
    }

    @Test
    @DisplayName("test executes only on Saturday")
    public void testAssumeTrueSaturday() {
        LocalDateTime dt = LocalDateTime.now();
        assumeTrue(dt.getDayOfWeek().getValue() == 6);
        System.out.println("further code will execute only if above assumption holds true");
    }

    @Test
    void testAssumeFalse() {
        boolean b = 'A' != 'A';
        assumeFalse(b);
        assertEquals("Hello", "Hello");
    }

    @Test
    @DisplayName("test executes only on Thursday")
    public void testAssumeTrueThursday() {
        LocalDateTime dt = LocalDateTime.now();
        assumeTrue(dt.getDayOfWeek().getValue() == 4);
        System.out.println("further code will execute only if above assumption holds true");
    }
}
```

运行结果：



4.2.4 JUnit5 测试异常（Test Exception)

在某些情况下，期望方法在特定条件下引发异常。 如果给定方法未引发指定的异常，则 assertThrows 将使测试失败。

JUnit5 assertThrows()的语法：

它断言所提供的 executable 的执行将引发 expectedType 的异常并返回该异常。

public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)

代码示例：

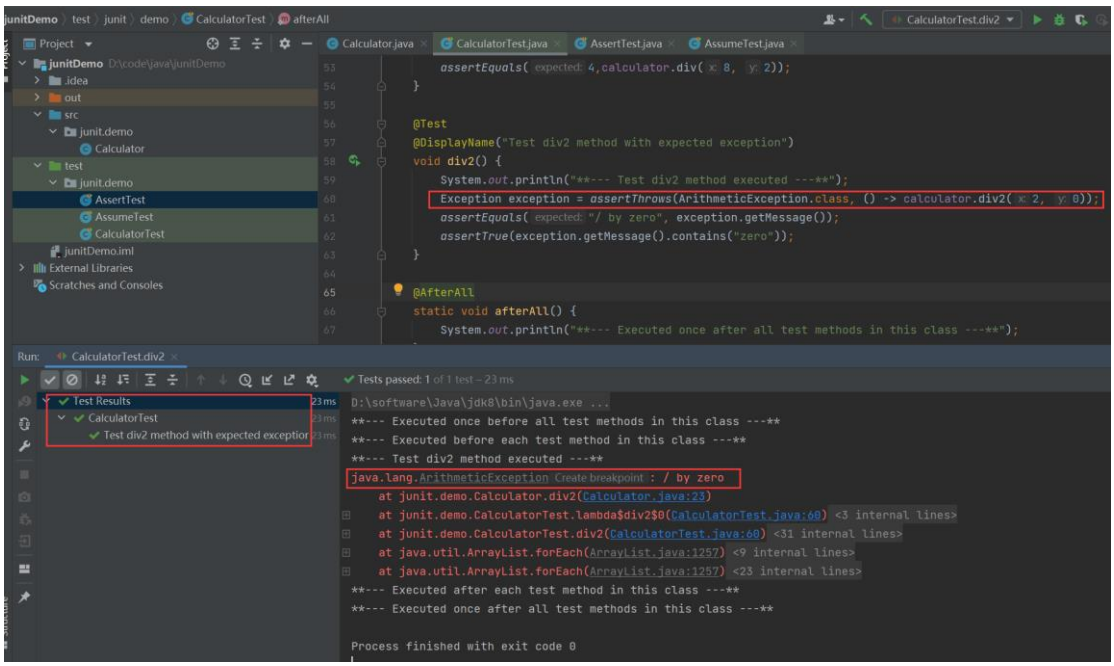
被测方法：

```
   public int div2(int x, int y) { //除法  做了异常判断
            try {
                  int z = x / y;
            } catch (Exception e) {
                  e.printStackTrace();
            }
            return x / y;
}
```

测试方法：

```
@Test
@DisplayName("Test div2 method with expected exception")
void div2() {
      System.out.println("**--- Test div2 method executed ---**");
      Exception exception = assertThrows(ArithmeticException.class, () -> calculator.div2(2, 0));
      assertEquals("/ by zero", exception.getMessage());
      assertTrue(exception.getMessage().contains("zero"));
}
```

运行结果：



4.2.5 JUnit5 参数测试（Test Exception)

要使用 JUnit 5 进行参数化测试，除了 junit-jupiter-engine 基础依赖之外，还需要另个模块依赖：junit-jupiter-params，其主要就是提供了编写参数化测试 API。@ParameterizedTest 作为参数化测试的必要注解，替代了 @Test 注解。任何一个参数化测试方法都需要标记上该注解。

（1）基本数据源测试： @ValueSource

@ValueSource 是 JUnit 5 提供的最简单的数据参数源，支持 Java 的八大基本类型、字符串和 Class，使用时赋值给注解上对应类型属性，以数组方式传递。

（2）CSV 数据源测试：@CsvSource

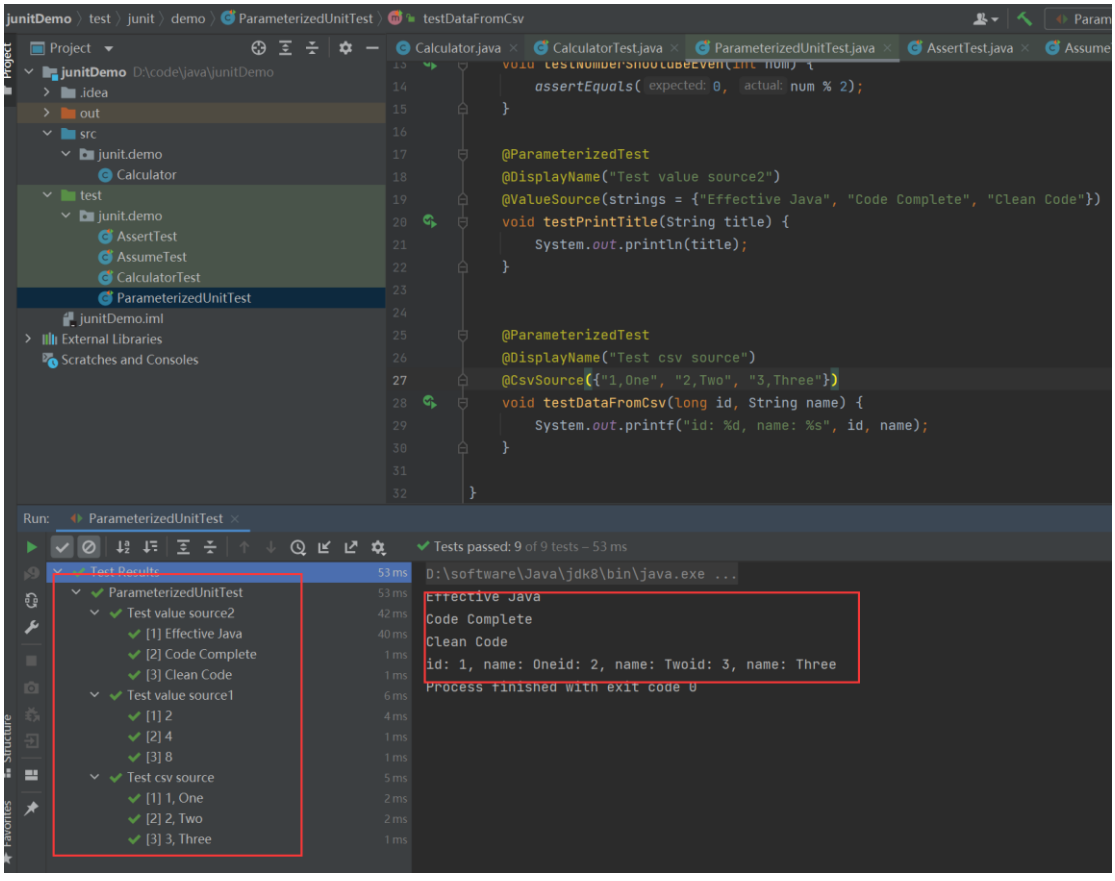
通过 @CsvSource 可以注入指定 CSV 格式 (comma-separated-values) 的一组数据，用每个逗号分隔的值来匹配一个测试方法对应的参数。

代码示例：

```
package junit.demo;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.ValueSource;
import static org.junit.jupiter.api.Assertions.*;

public class ParameterizedUnitTest {
    @ParameterizedTest
    @DisplayName("Test value source1")
    @ValueSource(ints = {2, 4, 8})
    void testNumberShouldBeEven(int num) {
```

```
        assertEquals(0, num % 2);
    }

    @ParameterizedTest
    @DisplayName("Test value source2")
    @ValueSource(strings = {"Effective Java", "Code Complete", "Clean Code"})
    void testPrintTitle(String title) {
        System.out.println(title);
    }


    @ParameterizedTest
    @DisplayName("Test csv source")
    @CsvSource({"1,One", "2,Two", "3,Three"})
    void testDataFromCsv(long id, String name) {
        System.out.printf("id: %d, name: %s", id, name);
    }

}
```

运行结果：



4.3 设计测试用例，编写单元测试代码

结合飞机大战实例，在系统中选择英雄机、敌机、子弹和道具类的方法（包含其父类方法）作为单元测试的对象，使用 4.2 中介绍的 JUnit5 的常用方法，为每个测试对象编写单元测试代码。要求至少选择 3 个类，每个类至少 2 个方法。

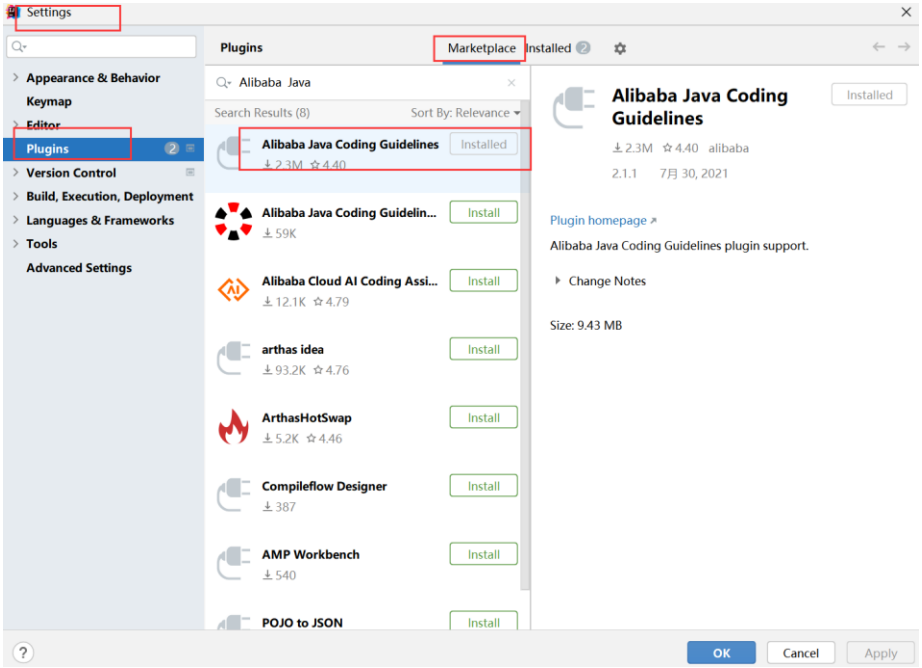
4.4 编码规范

无规矩不成方圆，无规范不能协作。编码规范是程序编码所要遵循的规则，要注意代码的正确性、稳定性、可读性。好的编码规范是提高我们代码质量的最有效的工具之一。要避免使用不易理解的数字，用有意义的标识来替代，不要使用难懂的技巧性很高的语句。

IntelliJ IDEA 提供多种插件帮助我们规范代码。以下介绍阿里编码规约插件（Alibaba Java Coding Guidelines）的安装和使用方法。该插件是对《阿里巴巴 Java 开发规约》的一个延伸，它以一个 IDE 的插件存在，可以自动对手册中的 Java 不规范的问题进行提示。

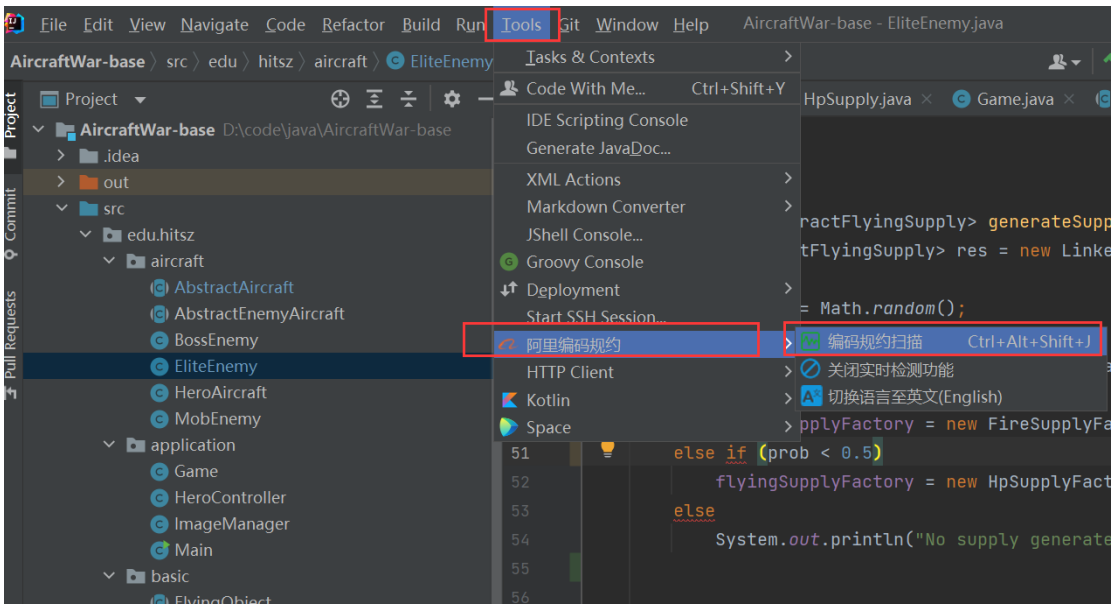
(1) Intellij IDEA 安装插件

通过 IDEA 的插件管理搜索 Alibaba 并集成安装后，重启 IDEA 即可。点击 File --> Settings --> Plugins --> Marketplace，搜索 Alibaba 。



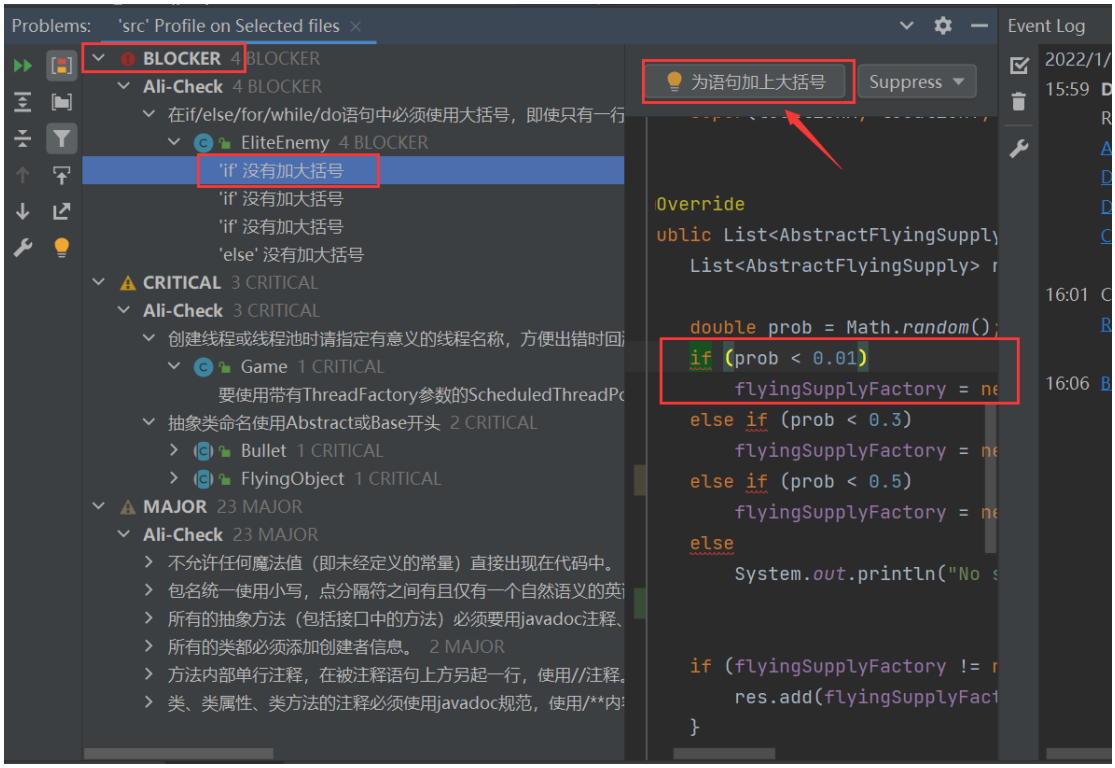
(2) 编码规范扫描

选中某一个类，或者在这个类里边右键，或者选择 Tools --> 阿里编码规约 --> 编码规约扫描。



(3) 扫描坏代码结果

扫描代码后，将不符合规约的代码按 Blocker（崩溃）/Critical（严重）/Major（重要） 三个等级显示在下方，双击可以定位至代码处，右侧窗口还有针对代码的批量修复功能。在实际工作中，Blocker/Critical 是必须要处理掉的。我们这里建议大家处理掉全部 Blocker/Critical 级别的问题。



5. 实验要求

1. 提交单元测试代码，代码运行正确；
2. 提交测试报告，内容包括所设计的单元测试用例描述及相应的单元测试结果截图；
3. 在测试报告中列举 2 个用阿里编码规约插件扫描出来的问题，并描述如何解决（截图和文字描述）。