@Test, Assert

Аннотация: `org.junit.Test`

Класс: `org.junit.Assert`

Описание

Библиотека JUnit применяется для написания *юнит-тестов* — небольших единиц кода для автоматизированного тестирования отдельных компонентов программы в изоляции от других её компонентов.

JUnit поставляется в составе среды Eclipse, но не включается в проекты по умолчанию. Для добавления JUnit в проект нужно зайти в свойства зависимостей проекта (правой кнопкой на проект → *Build Path* → *Configure Build Path*) и на вкладке *Libraries* добавить библиотеку JUnit кнопкой *Add Library* → *JUnit*.

Основы написания тестов

Тестом может служить любой класс с методами без параметров, помеченными аннотацией `@Test`. У класса также должен быть конструктор без параметров (подойдёт также класс без явного конструктора; как вы помните, если в классе явно не объявлено ни одного конструктора, компилятор генерирует пустой `public`-конструктор без параметров). Если в классе есть хотя бы один метод с аннотацией `@Test`, Eclipse позволяет запустить этот класс как тест JUnit с помощью команды контекстного меню *Run As* → *JUnit Test*.

Например, мы могли бы начать писать класс, тестирующий библиотечный класс `ArrayList`, следующим образом:

```
public class TestArrayList {
    @Test
    public void testAdd() {
        List<String> list = new ArrayList<>();
        list.add("Hello");
        // ...
    }

    @Test
    public void testAddAll() {
        List<String> list = new ArrayList<>();
        list.addAll(Arrays.asList("Hello", "World"));
        // ...
    }
}
```

Проверки (assertions)

Для проверки того, что тестируемые классы и методы работают правильно, мы используем статический класс `org.junit.Assert`. После первой же непройденной проверки выполнение тестового метода обрывается, и соответствующий этому методу тест считается проваленным.

Все методы класса `Assert` являются статическими. Как правило, в тесты имеет смысл импортировать этот класс статически:

```
import static org.junit.Assert.*;
```

Кстати, Eclipse предлагает сделать это, если вызвать любой метод `assertXXXX` внутри метода, отмеченного как `@Test`.

Методов проверки существует много. Вот самые главные из них (в квадратные скобки заключён необязательный аргумент, задающий сообщение об ошибке при провале проверки). Все эти методы объявлены как `static void`.

`fail([String message])`	Немедленно выходит из теста и помечает его как проваленный.
`assertTrue([String message], boolean condition)`	Проверяет, что указанное условие выполняется. Проверка считается успешной, если `condition == true`.
`assertFalse([String message], boolean condition)`	Проверяет, что указанное условие не выполняется. Проверка считается успешной, если `condition == false`.
`assertNull([String message], Object object)`	Проверяет, что указанная ссылка на объект равна `null`.
`assertNotNull([String message], Object object)`	Проверяет, что указанная ссылка на объект не равна `null`.
`assertEquals([String message,] long expected, long actual)`	Проверяет на равенство два целочисленных значения. Проверка считается успешной, если `expected == actual`.
`assertEquals([String message,] double expected, double actual, double delta)`	Проверяет, что два значения с плавающей точкой отличаются друг от друга не более чем на `delta`. Иными словами, проверка считается успешной, если `Math.abs(expected - actual) <= delta`.
`assertEquals([String message,] Object expected, Object actual)`	Проверяет два объекта на равенство по значению. Проверка считается успешной, если `Objects.equals(expected, actual) == true`.
`assertSame([String message,] Object expected, Object actual)`	Проверяет, что две ссылки ссылаются на один и тот же объект. Проверка считается успешной, если `expected == actual`.
`assertNotSame([String message,] Object expected, Object actual)`	Проверяет, что две ссылки ссылаются на разные объекты. Проверка считается успешной, если `expected != actual`.

С помощью этих методов мы можем реализовать проверки в наших методах, тестирующих `add` и

`addAll`:

```
import static org.junit.Assert.*;

public class TestArrayList {
    @Test
    public void testAdd() {
        List<String> list = new ArrayList<>();

        // предусловия
        assertTrue(list.isEmpty());
        assertEquals(0, list.size());

        list.add("Hello");

        // постусловия
        assertFalse(list.isEmpty());
        assertEquals(1, list.size());
        assertEquals("Hello", list.get(0));
    }

    @Test
    public void testAddAll() {
        List<String> list = new ArrayList<>();

        // предусловия
        assertTrue(list.isEmpty());
        assertEquals(0, list.size());

        list.addAll(Arrays.asList("Hello", "World"));

        // постусловия
        assertFalse(list.isEmpty());
        assertEquals(2, list.size());
        assertEquals("Hello", list.get(0));
        assertEquals("World", list.get(1));
    }
}
```

Важно! В семействах методов `assertEquals` и `assert[Not]Same` эталонное значение, с которым производится сравнение, передаётся первым параметром, а проверяемое значение — вторым. Логика теста будет работать при любом порядке параметров, но правильный порядок нужен для правильных сообщений об ошибке при провале теста.

```
assertEquals(1, list.size());  // Правильно
assertEquals(list.size(), 1);  // Неправильно
```

Идиома "ожидаемое исключение" (expected exception)

Иногда тест должен проверять, что метод выбрасывает определённое исключение, и считаться проваленным, если выполнение метода завершается нормально. Например, при тестировании `ArrayList` нам нужно проверить, что выход за границы списка корректно выбрасывает

`IndexOutOfBoundsException`.

Если нам нужно только проверить, что метод завершился по выбросу исключения, мы можем добавить параметр `expected` к аннотации `@Test`:

```
@Test(expected = IndexOutOfBoundsException.class)
public void testOutOfBounds() {
    List<String> list = new ArrayList<>();
    list.get(1);
}
```

Но что, если нам нужно организовать несколько таких проверок внутри одного метода, либо подвергнуть проверкам объект исключения? На помощь приходит идиома "expected exception". Мы заключаем код, выбрасывающий ожидаемое исключение, в блок `try-catch` и заставляем тест провалиться при нормальном завершении блока `try`:

```
try {
    // тестируемый код
    fail();
} catch (ExpectedExceptionType expected) {
    // анализируем объект исключения
}
```

Например, таким образом мы можем в одном методе проверить объект `ArrayList` на выход за границы диапазона в обоих направлениях:

```
@Test
public void testOutOfBounds() {
    List<String> list = new ArrayList<>();
    list.add("Hello");
    list.add("World");
    assertEquals(2, list.size());

    list.get(0); // OK
    list.get(1); // OK

    try {
        list.get(2);
        fail();
    } catch (IndexOutOfBoundsException expected) {
        assertEquals("index: 2, size: 2", expected.getMessage());
    }

    try {
        list.get(-1);
        fail();
    } catch (IndexOutOfBoundsException expected) {
        assertEquals("index: -1, size: 2", expected.getMessage());
    }
}
```

Выполнение кода до и после тестов

Методы, помеченные аннотациями `@Before` и `@After`, будут выполнены один раз перед и, соответственно, после каждого теста:

```
@Before
public void before() {
    System.out.println("Вхожу в тест");
}

@After
public void after() {
    System.out.println("Выхожу из теста");
}
```

Это бывает полезно, чтобы инициализировать какие-то зависимости тестов, а после их завершения убраться за собой.

Если же нужно, чтобы метод вызывался один раз перед прогоном *всех* объявленных в классе тестов, или один раз после завершения всех тестов, такие методы нужно объявить статическими и навесить на них аннотацию `@BeforeClass` и, соответственно, `@AfterClass`:

```
@BeforeClass
public static void beforeClass() {
    System.out.println("Выполняю тесты TestArrayList");
}

@AfterClass
public static void afterClass() {
    System.out.println("Тесты TestArrayList выполнены");
}
```

Доступ к деталям реализации

Иногда тестам для своей работы недостаточно интерфейса класса, и им нужен доступ к деталям реализации класса, помеченным как `private`.

Хорошим тоном считается размещать тесты в том же пакете, что и тестируемый класс. В этом случае для доступа к деталям реализации допустимо повысить видимость методов с `private` до `package-private`, а для полей объявить методы доступа с видимостью `package-private`. Конечно, такое решение следует задокументировать в комментариях, подчеркнув, что эти методы нужны только для тестов.

```
private SomeType someField;

// accessor for testing
SomeType getSomeField() { return someField; }

// visible for testing
void doSomethingNastyWithThisObject() {
    ...
}
```

Вот пример, где это может понадобиться. Допустим, нас чем-то не устраивает стандартный класс `String` (а зря!) и мы реализуем собственный строковой класс. Внутри у него, как и у стандартного класса

`String`, хранится массив `char`, и нам нужно проверить, что при вызове конструктора с параметром типа "массив `char`" он создаёт копию этого массива, а не хранит ссылку на переданный массив (что позволило бы изменить состояние неизменяемого объекта).

```
public class MyString {
    private final char[] value;

    public MyString(char[] value) {
        this.value = Arrays.copyOf(value, value.length);
    }
}

public class TestMyString {
    @Test
    public void testArrayCopied() {
        char[] value = { 'H', 'e', 'l', 'l', 'o' };
        MyString str = new MyString(value);
        // assertNotSame(value, ?)
    }
}
```

Мы не можем напрямую доступиться к полю `value`, потому что оно объявлено как `private`. Чтобы обратиться к его значению в тесте, нам нужно добавить для него геттер с видимостью `package-private` — специально для теста:

```
public class MyString {
    private final char[] value;

    public MyString(char[] value) {
        this.value = Arrays.copyOf(value, value.length);
    }

    // accessor for testing
    char[] getValue() { return value; }
}
```

Тогда тест записывается прямолинейно:

```
public class TestMyString {
    @Test
    public void testArrayCopied() {
        char[] value = { 'H', 'e', 'l', 'l', 'o' };
        MyString str = new MyString(value);
        assertNotSame(value, str.getValue());
    }
}
```

Если в проекте используется библиотека Google Guava, вместо комментария можно использовать аннотацию `@VisibleForTesting`, предназначенную как раз для такого случая.

```
@VisibleForTesting
char[] getValue() { return value; }
```

Наборы тестов

При наличии нескольких классов-тестов можно объединить их в *набор тестов* (test suite), который можно использовать, чтобы прогнать все тесты, включённые в набор. Для этого нужно создать новый класс, который можно даже оставить пустым, и пометить его аннотациями `@RunWith(Suite.class)` и `@SuiteClasses`:

```
@RunWith(Suite.class)
@SuiteClasses({
    TestArrayList.class,
    TestMyString.class,
})
public class CoreUtilsTestSuite { }
```

Такой класс точно так же можно запустить из Eclipse с помощью *Run As → JUnit Test*.