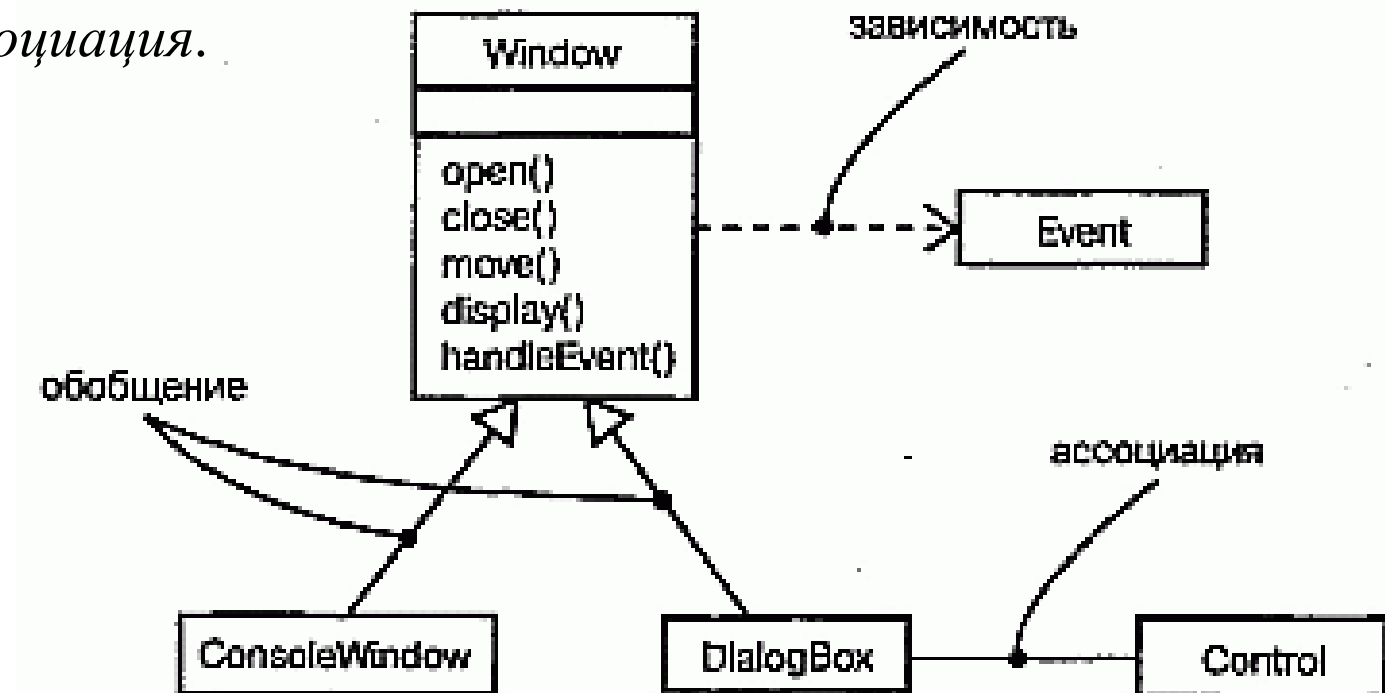


Типы отношений между классами и интерфейсами

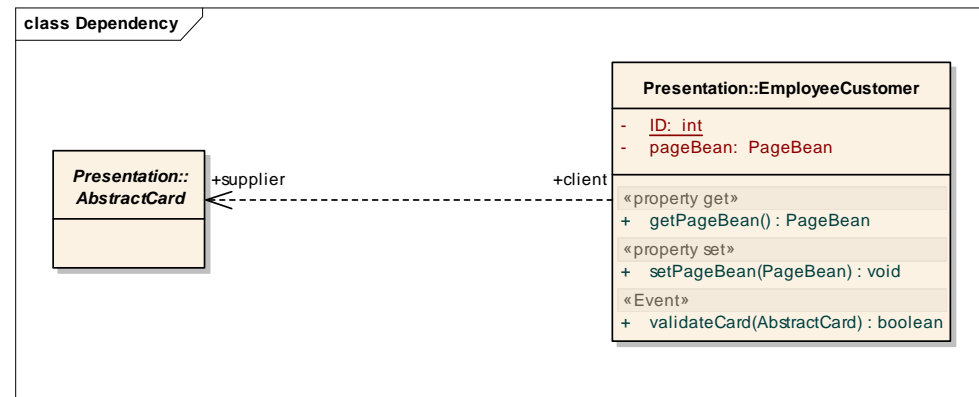
Типы отношений между классами

- ❑ **Отношения** классов/интерфейсов – это связи, отображаемые различными линиями между классами/интерфейсами.
- ❑ Выделяют четыре типа наиболее важных отношений:
 - *зависимость*,
 - *обобщение*,
 - *реализация*
 - *ассоциация*.



Типы отношений между классами

- ❑ **Зависимость** (Dependency) называется отношение использования, определяющее, что изменение состояния объекта одного класса может повлиять на объект другого класса, который его использует, причем обратное в общем случае неверно (применяются тогда, когда экземпляр одного класса использует экземпляр другого, *например*, в качестве параметра метода).

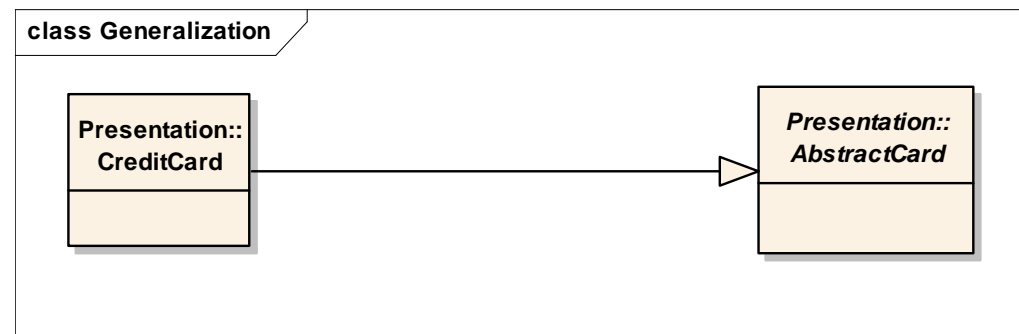


- ❑ Графически зависимость изображается пунктирной линией с открытой стрелкой, направленной к тому классу, от которого зависит другой класс.

Например, Стипендия зависит от Экзамена.

Типы отношений между классами

- ❑ **Обобщение** (Generalization) - это отношение между общей сущностью (суперклассом) и ее конкретным воплощением (подклассом), т.е. объекты класса-потомка могут использоваться всюду, где встречаются объекты класса-родителя, но не наоборот.

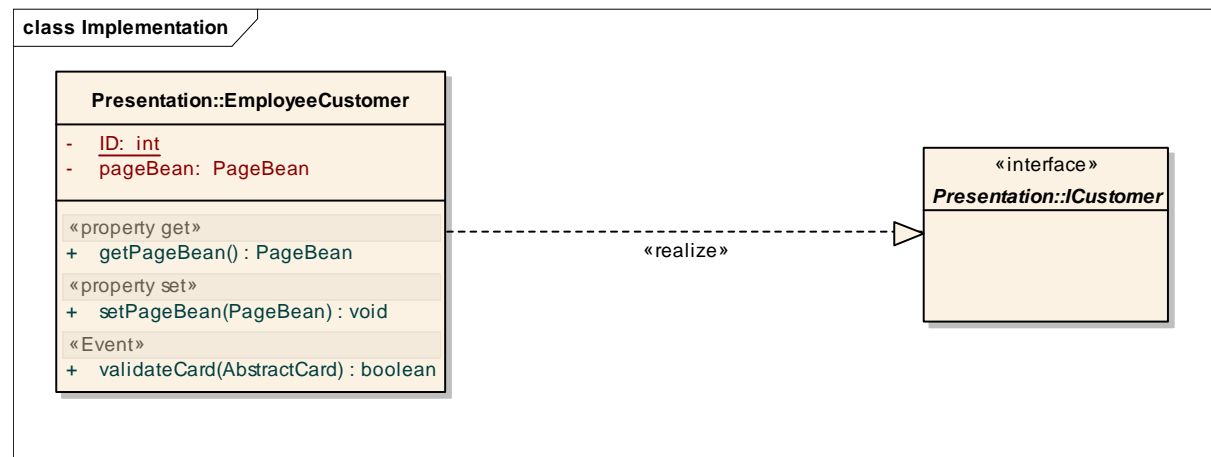


- ❑ Обобщения иногда называют отношениями типа **"is-a"** ("является разновидностью").
- ❑ Графически обобщение изображается сплошной линией с закрытой стрелкой, направленной к суперклассу.

Например, с **Potato** (Картофель) - это Овощ, **Bus** (Автобус) - это Транспортное средство и т.д.

Типы отношений между классами

- ❑ **Реализацией** (Realization) называется отношение между классификаторами (классами, интерфейсами), при котором один описывает контракт (интерфейс сущности), а другой гарантирует его выполнение.

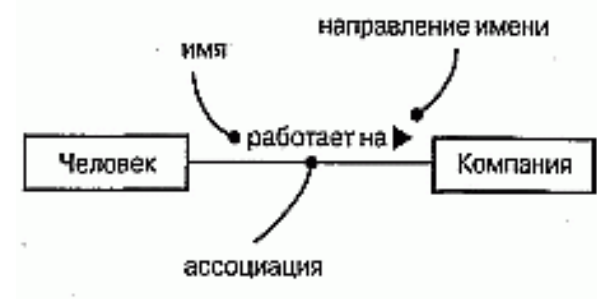


- ❑ Графически реализация изображается пунктирной линией с закрытой стрелкой, направленной к интерфейсу.

Например, **PrinterSetup** (настройки принтера) реализуются принтерами **MatrixPrinter** и **LaserPrinter**.

Типы отношений между классами

- ❑ **Ассоциация** (Association) показывает, что объект одного класса связан с объектом другого класса и отражает некоторое отношение между ними.

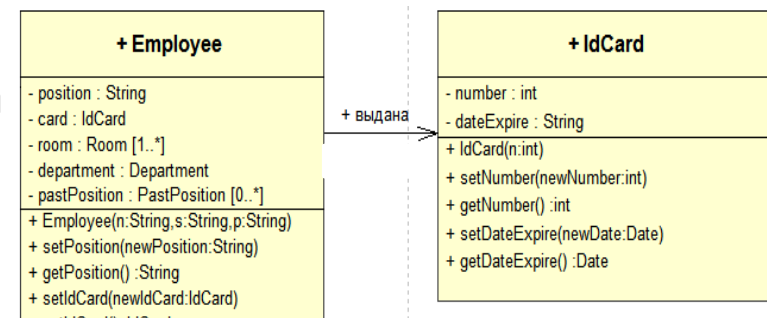


- ❑ Графически изображается сплошной линией между двумя классами.

Например: Студент учится на Факультете, Студент учится у Преподавателя.

- ❑ Ассоциация может быть направленной:

Например, каждому сотруднику может соответствовать только одна идентификационная карточка



Типы отношений между классами

- ❑ **Множественность** (Multiplicity, кратность) является деятельной логической ассоциацией, когда отображается мощность класса по отношению к другим классам.

Например, один авиационный парк может включать несколько самолетов, а один самолет может перевозить множество пассажиров или не одного.

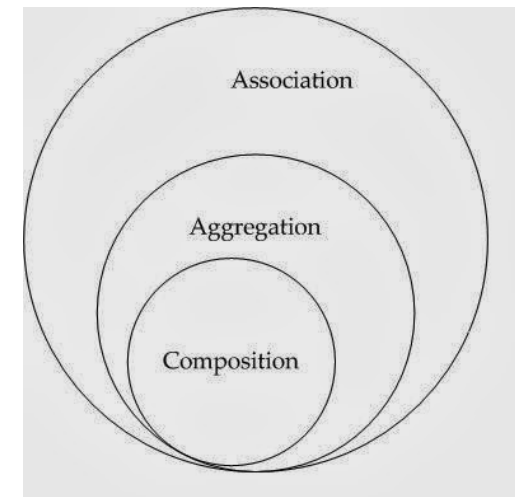


- ❑ Кратность может быть: один-ко-многим, ни-одного-ко-многим, многие-к-одному, многие-ко-многим и т.д.

нотация	объяснение	пример
0..1	Ноль или один экземпляр	кошка имеет или не имеет хозяина
1	Обязательно один экземпляр	у кошки одна мать
0..* или *	Ноль или более экземпляров	у кошки может быть, а может и не быть котят
1..*	Один или более экземпляров	у кошки есть хотя бы одно место, где она спит

Типы отношений между классами

- ❑ Простая ассоциация между двумя классами отражает структурное отношение между равноправными сущностями, когда оба класса находятся на одном концептуальном уровне и ни один не является более важным, чем другой.
- ❑ Иногда приходится моделировать отношение типа "часть/целое", в котором один из классов имеет более высокий ранг (целое) и состоит из нескольких меньших по рангу (частей).
- ❑ Вариантами таких отношений ассоциации являются:
 - *агрегация*
 - *композиция*.



Типы отношений между классами

- ❑ **Агрегация** (Aggregation) - формирование определенного класса как результата объединения или построения коллекции из объектов других классов или его собственного.



- ❑ Графически изображается сплошной линией с не закрашенным ромбом со стороны "целого". Может быть направленной.

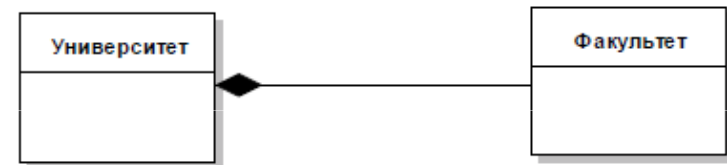
Например, Студент входит в Группу любителей java. Однако этот же студент может входить и в другие группы тоже.

- ❑ Агрегация может рассматриваться как отношение "**has_a**" ("имеет в своем составе");
- ❑ Вложенные объект (часть) может выжить или существовать без окружающего класса.

Типы отношений между классами

- ❑ Ограниченная агрегация называется **композицией** (Composition): член объекта (часть) не может существовать без содержащего класса (целого, контейнера);

Например, **Собака** имеет **Хвост**, но **Хвост** не может существовать без **Собаки**.



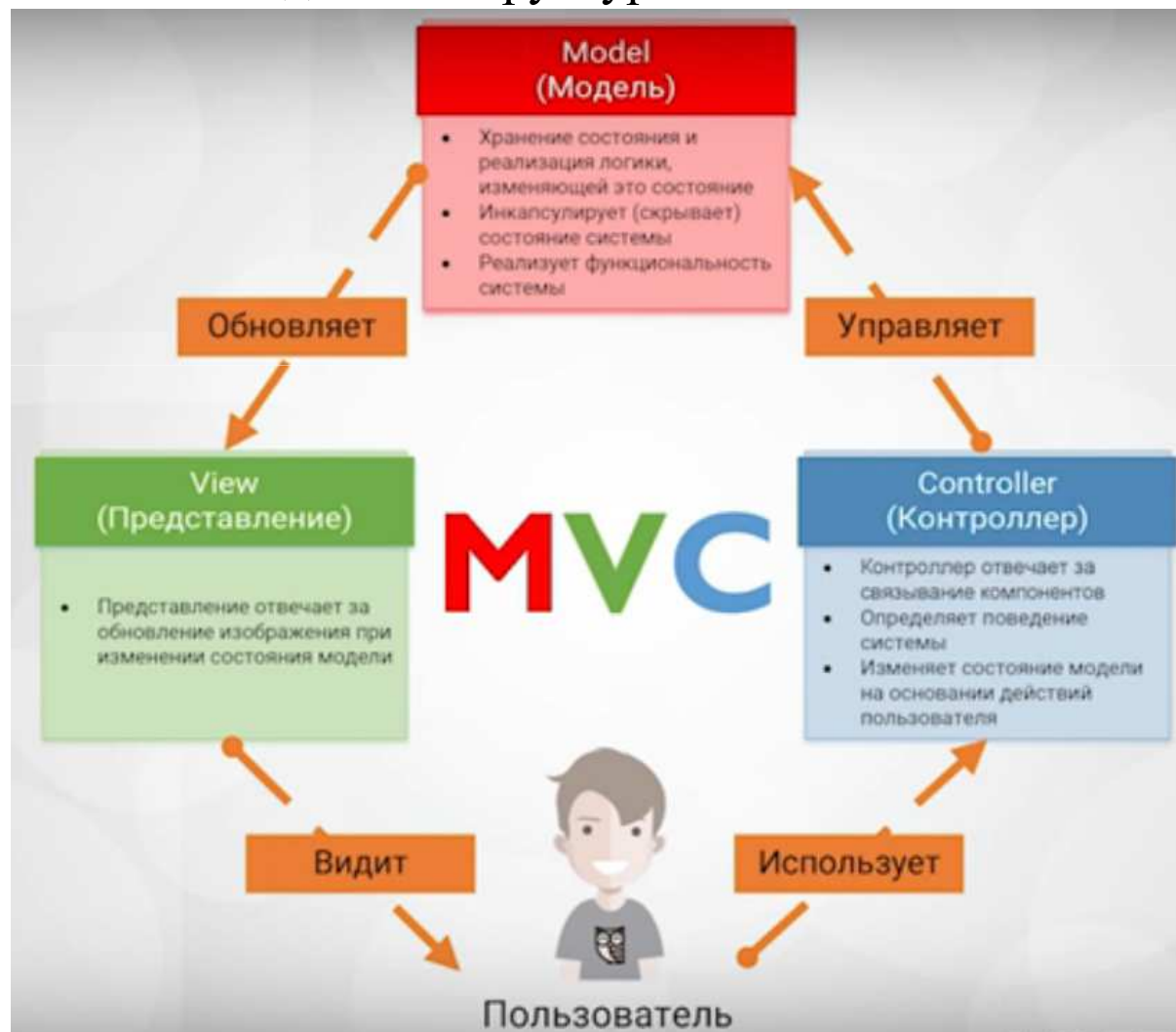
- ❑ Графически изображается сплошной линией, соединяющей два класса, с заполненным ромбом, прилегающим к классу контейнеру и открытой стрелкой на содержащийся класс.



Архитектурный шаблон MVC

Архитектурный шаблон MVC

- MVC — это шаблон, который описывает способ построения структуры приложения, сферы ответственности и взаимодействие каждой из частей в данной структуре.



Архитектурный шаблон MVC

Пример 1: *Модель*

```
package com.epam.model;

public class CalculateModel {
    private int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public void incrementValue() {
        ++value;
    }
}
```

Архитектурный шаблон MVC

Продолжение примера 1: Представление

```
package com.epam.view;

public class CalculateView {

    public static final String INPUT_INT_DATA =
        "Enter an integer = ";
    public static final String WRONG_INPUT_INT_DATA =
        "Wrong input! Repeat please! ";
    public static final String OUR_INT = "Increment result = ";
    public void printMessage(String message) {
        System.out.print(message);
    }

    public void printMessageAndResult(String message, int value) {
        System.out.println(message + value);
    }
}
```

Архитектурный шаблон MVC

Продолжение примера 1: Контроллер

```
package com.epam.controller;

import com.epam.model.CalculateModel;
import com.epam.view.CalculateView;
import com.epam.service.InputUtility;

public class CalculateController {
    private CalculateModel model;
    private CalculateView view;
    public CalculateController(CalculateModel model,
                               CalculateView view) {
        this.model = model;    this.view = view;
    }
    public void calculate() {
        model.setValue(InputUtility.inputIntValueWithScanner(view));
        model.incrementValue();
        view.printMessageAndResult(view.OUR_INT,
                                   model.getValue());
    }
}
```


Архитектурный шаблон MVC

Продолжение примера 1: Утилитный класс

```
import com.epam.view.CalculateView;
import java.util.Scanner;

public class InputUtility {
    private static Scanner sc = new Scanner(System.in);
    public static int inputIntValueWithScanner(
                                                CalculateView view) {
        view.printMessage(view.INPUT_INT_DATA);
        while( !sc.hasNextInt()) {
            view.printMessage(view.WRONG_INPUT_INT_DATA +
                                view.INPUT_INT_DATA);
            sc.next();
        }
        return sc.nextInt();
    }
}
```

Архитектурный шаблон MVC

Продолжение примера 1: Запуск программы

```
import com.epam.controller.CalculateController;
import com.epam.model.CalculateModel;
import com.epam.view.CalculateView;

public class MVCMain {
    public static void main(String[] args) {
        CalculateView view = new CalculateView();
        CalculateModel model = new CalculateModel();
        CalculateController controller =
                                new CalculateController(model, view);
        controller.calculate();
    }
}
```

Вывод в консоли:

```
Wrong input! Repeat please! Enter an integer = 12.7
Wrong input! Repeat please! Enter an integer = 44
Increment result = 45
```



Модульное тестирование

Модульное тестирование

- ❑ **Модульное тестирование** (*unit testing*) – это этап в разработке ПО, позволяющий проверить на корректность отдельные модули исходного кода программы (идея - написать тесты для каждой нетривиальной функции или метода):
 - позволяет достаточно быстро проверить, не привело ли очередное изменение кода к появлению ошибок в уже написанных и протестированных местах программы;
 - облегчает локализацию и устранение таких ошибок;
- ❑ Разработка через тестирование – это процесс разработки программного обеспечения, который предусматривает написание и автоматизацию модульных тестов еще до момента написания соответствующих классов или модулей:
 - гарантирует, что все обязанности любого элемента программного обеспечения определяются еще до того, как они будут закодированы.

Цели

❑ Поощрение изменений

- позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений);

❑ Упрощение интеграции

- помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируются отдельные части программы, затем программа в целом;

❑ Документирование кода

- тесты можно рассматривать как «живой документ» для тестируемого класса (клиенты, которые не знают, как использовать данный класс, могут использовать тест в качестве примера).

Модульное тестирование

□ Отделение интерфейса от реализации

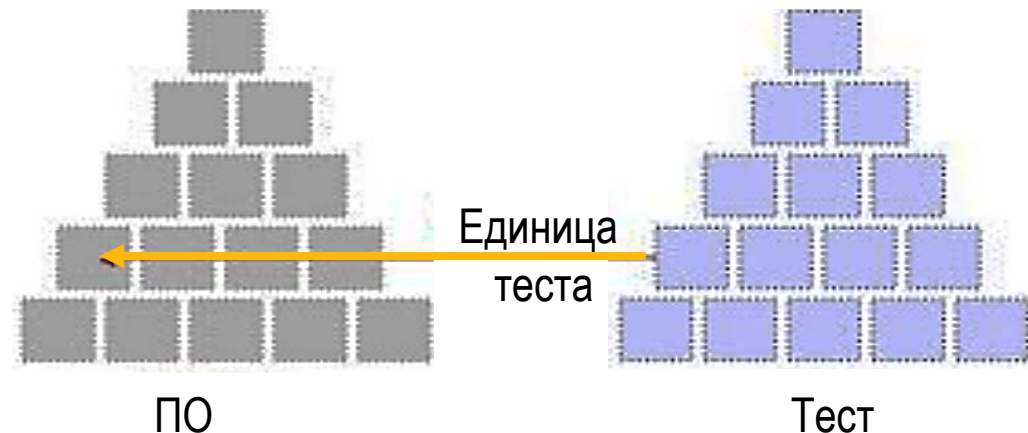
- поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним:
 - ✓ *например*, класс пользуется базой данных. В ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный mock-объект (автоматически генерируемые заглушки, которые могут выступать в роли реальных объектов). Это приводит к менее связанному коду, минимизируя зависимости в системе

□ Баг-трэкинг

- В случае обнаружения бага для него можно (даже рекомендуется) создать тест для выявления повторения подобной ошибочной ситуации при последующем изменении кода.

Модульное тестирование

- ❑ Для организации модульного тестирования в Java используется семейство фреймворков **Junit**.
- ❑ Правила, которым все фреймворки модульного тестирования должны следовать:
 - модульный тест проверяет поведение отдельной единицы работы;
 - отдельная единица работы - часто (но не всегда) один метод;
 - каждый модульный тест должен работать независимо от других модульных тестов;
 - фреймворк должен обнаруживать и сообщать ошибки тест за тестом.



Модульное тестирование

❑ Версии:

- Junit – до Java 1.5.0 (наследуем и расширяем классы);
- JUnit 4 – с Java 1.5.0 (используем аннотации).

❑ Соглашения: Для JUnit

- имя любого метода, предназначенного для функционирования в качестве логического теста, начинается с префикса **test** (любой метод, имя которого начинается с этого префикса, *например*, **testUserCreate**, выполняется в соответствии с хорошо описанным процессом тестирования, который гарантирует исполнение соответствующей фикстуры (fixture) как до, так и после этого тестового метода);
- класс, содержащий тесты, должен являться расширением класса **TestCase** среды JUnit (или некоторым производным от него).

Модульное тестирование

Пример 6, до JUnit4:

```
public class ClassToTest {  
    static public int increment(int a) {  
        return a++;  
    }  
}
```

...

```
public static void main(String[] args) {  
    TestRunner runner = new TestRunner();  
    TestSuite suite = new TestSuite();  
    suite.addTest(new TestClassToTest ("testIncrement"));  
    runner.run(suite);  
}
```

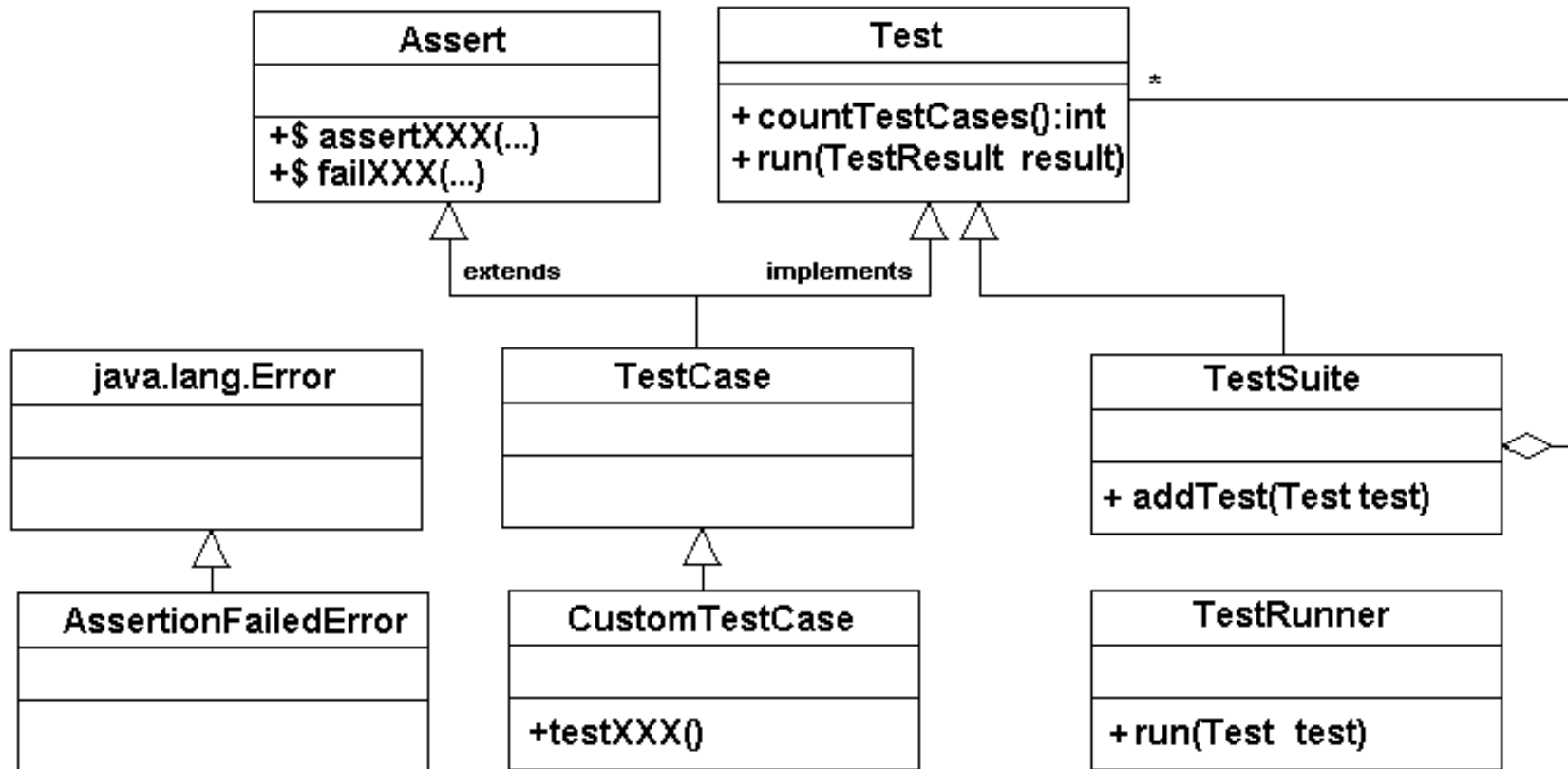
...

```
import junit.framework.*;
```

```
public class TestClassToTest extends TestCase {  
    public TestClassToTest (String name) {  
        super(name);  
    }  
    public void testIncrement() {  
        int result = ClassToTest.increment(2);  
        assertEquals(result , 3);  
    }  
}
```

Модульное тестирование

UML-диаграмма применения фреймворка JUnit:



<http://junit.sourceforge.net/javadoc/junit/framework/TestCase.html>

<http://junit.sourceforge.net/javadoc/junit/framework/Assert.html>

<http://junit.sourceforge.net/javadoc/org/junit/Test.html>

Модульное тестирование

❑ **Assert** (диагностика) - предназначен для сверки реального состояния тестируемого кода с ожидаемым.

java.lang.Object

| +--junit.framework.Assert

- assertTrue
- assertFalse
- assertEquals
- assertNull
- assertNotNull
- assertEquals

java.lang.Object

| +--java.lang.Throwable

| +--java.lang.Error

| +--junit.framework.AssertionFailedError

Модульное тестирование

Пример 7, JUnit – тестирование исключений

```
...  
public class TestException extends TestCase {  
...  
    public void testException() throws Exception {  
        try{  
            unsafeCall(...);  
            // Test Fail  
            fail("No exception was thrown");  
        } catch(OurException e) {  
            // Test OK  
        }  
    }  
...  
}
```

❑ **Фикстура (Fixture)** – это состояние среды тестирования, которое требуется для успешного выполнения тестового метода:

- может быть набор каких-либо объектов, состояние базы данных, наличие определенных файлов и т.д.;
- создается в методе ***setUp()*** перед каждым вызовом метода вида ***testSomething*** теста (**TestCase**);
- удаляется в методе ***tearDown()*** после окончания выполнения тестового метода.

```
public class TestClassToTest extends TestCase {  
    // will run before test execution  
    protected void setUp() throws Exception {  
        ...  
    }  
    // will run after test execution  
    protected void tearDown() throws Exception {  
        ...  
    }  
}
```

Фреймворк JUnit 4

- ❑ В JUnit 4 за счет использования аннотаций удалось полностью отказаться обоих вышеуказанных соглашений:
 - отпадает необходимость в иерархии классов;
 - методы, предназначенные для функционирования в качестве тестов, достаточно промаркировать новой аннотацией: **@Test**
- ❑ JUnit 4 отказывается от понятия «ошибка»:
 - предшествующие версии JUnit сообщали и о количестве *неудач* и о количестве *ошибок* (в версии JUnit 4 тест или проходит *успешно*, или завершается *неудачей*).

```
import org.junit.Test;
public class TestClassToTest {
    @Test
    public void increment() {
        ....
    }
}
```


Модульное тестирование

❑ В JUnit 4 не нужно использовать блоки **try-catch**:

- достаточно объявить ожидаемое исключение в аннотации **@Test**, т.е.

@Test(expected = Exception.class) - проверяет, выбрасывает ли метод указанное исключение;

```
public class TestClassToTest {  
    @Test(expected=OurException.class)  
    public void testException() {  
        unsafeCall(...);  
    }  
}
```

- ❑ Выполнение некоторых unit-тестов может занимать больше времени, чем у нас есть (*например*, тест требует соединения с внешним асинхронным ресурсом):
 - все что нужно сделать – это указать параметр ***timeout*** с необходимым значением в аннотации ***@Test***;
 - если максимальное отведенное тесту время истекает, то мы получаем сообщение об ошибке и о не выполнении теста
(*например*, `java.lang.Exception: test timed out after 5000 milliseconds`).

```
@Test(timeout=5000)  
public void increment() {  
    ...  
}
```

Модульное тестирование

- ❑ Поскольку методы *setUp()* и *tearDown()* упразднены, то необходимые для инициализации и освобождения ресурсов методы мы маркируем помощью аннотаций **@Before** или **@After**.
- ❑ Можно промаркировать несколько методов как **@Before** или **@After** (*порядок их вызова может быть любой – решает среда исполнения*).

```
public class TestClassToTest {  
  
    @Before  
    public void prepareTestData() { ... }  
  
    @Before  
    public void setupConnection() { ... }  
  
    @After  
    public void freeConnection() { ... }  
}
```

Модульное тестирование

- ❑ Можно создать метод, который будет выполняться только один раз перед исполнением любых тестовых методов в классе, используя аннотацию **@BeforeClass**:
 - метод должен быть **public static void** и без параметров.

```
@BeforeClass
```

```
public static void beforeClass() { ... }
```

- ❑ Имеет смысл использовать эту аннотацию для теста в случае:
 - когда класс содержит несколько тестов, использующих различные предустановки;
 - когда несколько тестов используют одни и те же данные, чтобы не тратить время на их создание для каждого теста.

Модульное тестирование

- ❑ Можно создать метод, который будет выполняться только один раз после всех тестов в классе, что были выполнены, используя аннотацию **@AfterClass**:
 - метод должен быть **public static void** и без параметров.

```
@AfterClass  
public static void afterClass() { ... }
```

- ❑ Имеет смысл использовать эту аннотацию для теста в случае:
 - когда использовалась аннотация **@BeforeClass**.

Модульное тестирование

- ❑ В некоторых ситуациях может понадобиться отключить некоторые тесты. *Например:*
 - если исходный код был изменен, а тест еще не был адаптирован;
 - тест постоянно валится и его исправление отложено до «светлого будущего».
- ❑ Для этого применяется аннотация **@Ignore**.

```
public class TestClassToTest {  
  
    @Ignore("Not running because <reason here>")  
    @Test  
    public void increment() {  
        ...  
    }  
}
```

Модульное тестирование

- ❑ Так как больше нет наследования от **TestCase**, но все еще нужны методы *assert...()*, то необходимо использовать статический импорт.

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

public class TestClassToTest {
    @Test
    public void increment() {
        ...
        assertEquals( result, 3 );
    }
}
```

- ❑ Диагностический метод сравнивает фактическое значение, возвращаемое тестом, с ожидаемым значением, и бросает **AssertionException** если сравнение теста не удастся.

Методы класса Assert

- ❑ Тест проходит, если **Object** не является нулевым (null):

assertNotNull("message", obj);

- ❑ Тест проходит, если **Object** является нулевым (null) :

assertNull("message", obj);

- ❑ Тест проходит, если два данные тождественны:

assertEquals("message", expected, actual);

- ❑ Тест проходит, если условие **true**:

assertTrue(true | false);

- ❑ Тест проходит, если два **Object** не один и тот же объект:

assertNotSame("message", expected, actual);

- ❑ Тест проходит, если два **Object** один и тот же объект :

assertSame("message", expected, actual);

Пример 8:

```
import static org.junit.Assert.*;  
import org.junit.Test;  
public class AssertionsTest {  
    @Test  
    public void test() {  
        String obj1 = "junit";  
        String obj2 = "junit";  
        String obj3 = "test";  
        String obj4 = "test";  
        String obj5 = null;  
        int var1 = 1;  
        int var2 = 2;  
        ...  
    }  
}
```

...

Продолжение примера 8:

...

```
int[] arr1 = { 1, 2, 3 };
```

```
int[] arr2 = { 1, 2, 3 };
```

```
assertEquals(obj1, obj2);
```

```
assertSame(obj3, obj4);
```

```
assertNotSame(obj2, obj4);
```

```
assertNotNull(obj1);
```

```
assertNull(obj5);
```

```
assertTrue(var1 != var2);
```

```
assertArrayEquals(arr1, arr2);
```

```
}
```

```
}
```

Модульное тестирование

Пример 9, класс, который будет тестироваться:

```
public class MyCalculate {  
    public int calcSum(int a, int b){  
        return a+b;  
    }  
    public int calcSub(int a, int b){  
        return a-b;  
    }  
    public int calcMult(int a, int b){  
        return a*b;  
    }  
    public int calcDiv(int a, int b){  
        return a/b;  
    }  
}
```

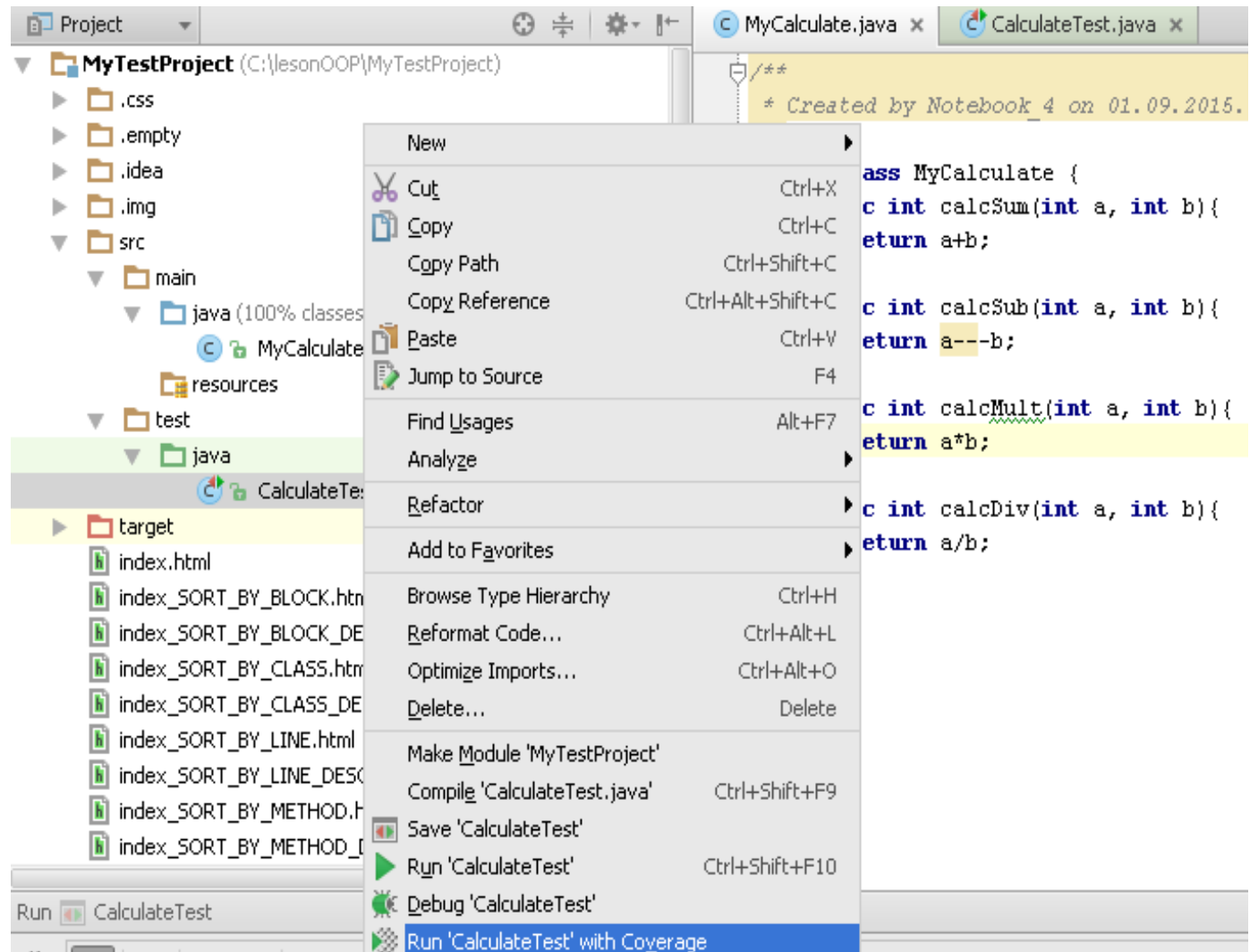
Модульное тестирование

Продолжение примера 9:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class CalculateTest {
    @Test
    public void testCalcSum() {
        MyCalculate testobj = new MyCalculate();
        assertEquals(4, testobj.calcSum(2, 2));
    }
}
```

Модульное тестирование

- ❑ Пример, как запустить тестовый класс с освещением работы в среде



Модульное тестирование

❑ Пример результата

Coverage Summary for Class: MyCalculate (<empty package name>)

Class	Class, %	Method, %	Line, %
MyCalculate	100% (1/ 1)	40% (2/ 5)	40% (2/ 5)

```
1  /**
2   * Created by Notebook_4 on 01.09.2015.
3   */
4  public class MyCalculate {
5      public int calcSum(int a, int b){
6          return a+b;
7      }
8      public int calcSub(int a, int b){
9          return a-b;
10     }
11     public int calcMult(int a, int b){
12         return a*b;
13     }
14     public int calcDiv(int a, int b){
15         return a/b;
16     }
17 }
```

Модульное тестирование

- Изменим тестовый класс в примере 9, дополнив тестовыми методами:

```
public class CalculateTest {    MyCalculate testobj;  
    @Before  
    public void initialize() {  
        testobj= new MyCalculate();  
    }  
    @Test  
    public void testCalcSum(){  
        assertEquals(4, testobj.calcSum(2, 2));  
    }  
    @Test  
    public void testCalcSub(){  
        assertEquals(0, testobj.calcSub(2, 2));  
    }  
    @Test  
    public void testCalcMult(){  
        assertEquals(4, testobj.calcMult(2, 2));  
    }  
}
```


Модульное тестирование

Coverage Summary for Class: MyCalculate (<empty package name>)

Class	Class, %	Method, %	Line, %
MyCalculate	100% (1/ 1)	80% (4/ 5)	80% (4/ 5)

```
1  /**
2   * Created by Notebook_4 on 01.09.2015.
3   */
4  public class MyCalculate {
5      public int calcSum(int a, int b){
6          return a+b;
7      }
8      public int calcSub(int a, int b){
9          return a-b;
10     }
11     public int calcMult(int a, int b){
12         return a*b;
13     }
14     public int calcDiv(int a, int b){
15         return a/b;
16     }
17 }
```

Модульное тестирование

- ❑ Проведем изменения в методе *calcSub()* класса **MyCalculate** и получим информации об ошибке:

```
public class MyCalculate {  
    public int calcSum(int a, int b){  
        return a+b;  
    }  
    public int calcSub(int a, int b){  
        return a-+-b;  
    }  
    public int calcMult(int a, int b){  
        return a*b;  
    }  
}
```

...

java.lang.AssertionError:
Expected :0
Actual :4

Параметризированный запуск

- ❑ Применятся для запуска одного и того же теста с разными входными данными;
- ❑ Для этого к классу тестов нужно добавить аннотацию **@RunWith** и указать в качестве параметра *value* значение **Parametrized.class**;
- ❑ Добавить в класс тестов метод генерации параметров с аннотацией **@Parametrized.Parameters**;
- ❑ Описать в методе генерации параметров сами параметры;
- ❑ Добавить в класс тестов конструктор для инициализации данных.

Аннотация **@RunWith**

- ❑ Фреймворк JUnit 4 имеет специальные классы – runner's, которые ответственны за то, как запускать тесты;
- ❑ С помощью аннотации **@RunWith** можно указать, какой runner's использовать;
- ❑ Для запуска тестов по своему сценарию можно реализовать свой runner и указать его в качестве параметра *value* для аннотации **@RunWith**.

Класс **Parametrized**

- ❑ Является реализацией runner'a, добавляя возможность использовать параметризованный запуск одних и тех же тестов.

Аннотация `@Parameterized.Parameters`

- ❑ Применяется для того, чтобы маркировать метод, используемый при создании набора данных для параметров;
- ❑ Метод должен возвращать коллекцию данных в виде массивов объектов, которые вы хотите использовать в качестве параметров;
- ❑ Метод должен быть объявлен как **public static**.

Например,

```
@Parameterized.Parameters
public static Collection data() {
    return Arrays.asList( new Object[][] {
        { 3, 2 }, // ожидаемое, параметр
        { 2, 3 }
    });
}
```

Передача параметров методам теста

- ❑ Для передачи значений параметров тестовым методам необходимо в классе тестов описать поля;
- ❑ Инициализировать описанные поля можно двумя способами:
 - создать конструктор класса и в нем реализовать присваивание значений параметров. *Например,*

```
@RunWith(value=Parameterized.class)
public class TestClassToTest {
    private int expected;
    private int value;

    public TestClassToTest(int expected, int value) {
        this.expected = expected;
        this.value = value;
    }

    // ....
}
```

Модульное тестирование

- маркировать соответствующие поля аннотацией `@Parameterized.Parameter`. Например,

```
@RunWith(value=Parameterized.class)
public class TestClassToTest {
    @Parameterized.Parameter(0)
    public int expected;
    @Parameterized.Parameter(1)
    public int value;

    // ....
}
```

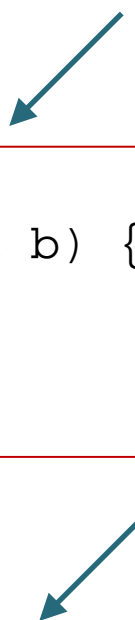


- ✓ можно указать индекс, по которому брать значение для этого поля из массива объектов;
- ✓ Поля должны быть описаны как **public**

Модульное тестирование

Пример 10:

Тестируемый
класс



```
public class Calculator {  
    public double sum(double a, double b) {  
        return (a + b);  
    }  
}
```

Класс тестов

```
@RunWith(Parameterized.class)  
public class TestCalculator {  
    @Parameterized.Parameter  
    public int firstParameter;  
    @Parameterized.Parameter(1)  
    public int secondParameter;  
    @Parameterized.Parameter(2)  
    public int expectedResult;  
  
    // ...  
}
```


Модульное тестирование

Продолжение примера 10:

Аннотация имеет параметр **name** со значением по умолчанию "index", для отображения номера строки данных из массива объектов

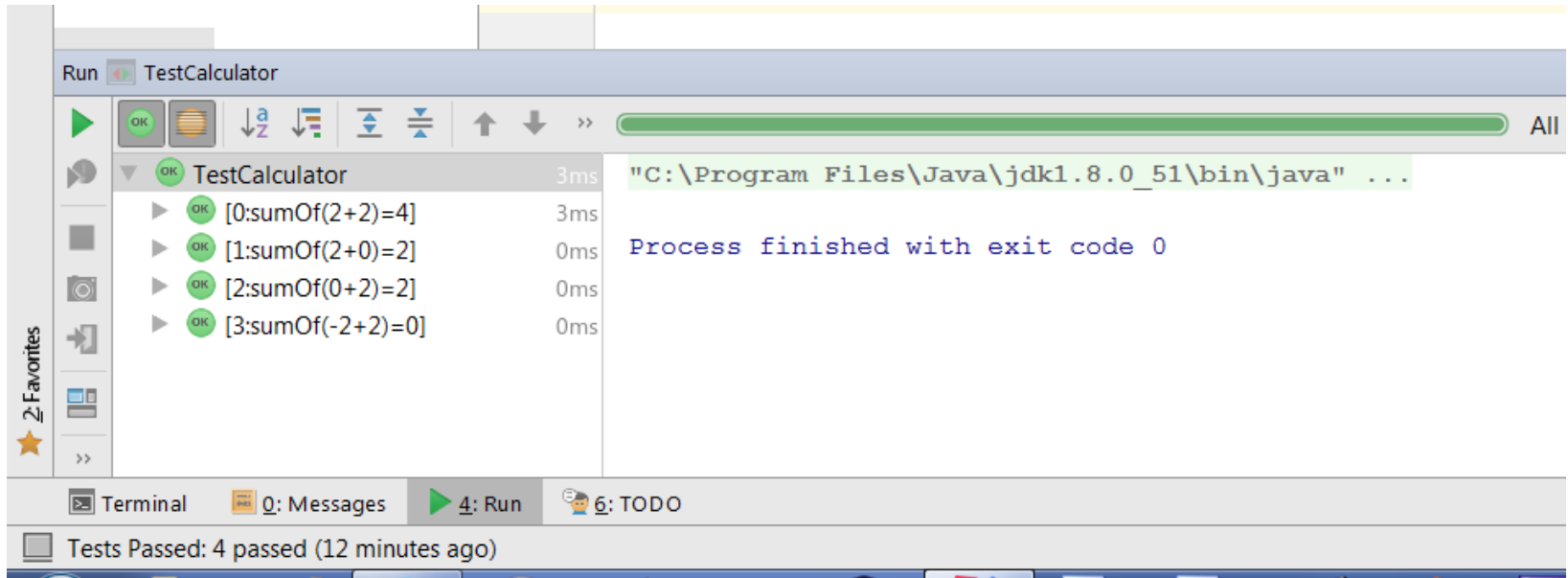
```
// ...
@Parameterized.Parameters(name="{index}:sumOf({0}+{1})={2}")
public static Collection<Object[]> getTestData() {
    return Arrays.asList(new Object[][]{
        {2, 2, 4},
        {2, 0, 2},
        {0, 2, 2},
        {-2, 2, 0}
    });
}

@Test
public void testSum() {
    Calculator calculator = new Calculator();
    double result = calculator.sum(firstParameter, secondParameter);
    Assert.assertEquals("Результат(" + result + ") не равен "
        + expectedResult, expectedResult, result, 0.001);
}
}
```

Параметр **name** дополнен информацией для отображения самих данных

Модульное тестирование

Результат выполнения тестов:





Блоки инициализации

Блоки инициализации

❑ Перед выполнением приложения Java *загрузчик классов* Java загружает его начальный класс – класс с методом ***public static void main(String [] args)***, – и верификатор Java проверяет байт-код этого класса. *Затем этот класс инициализируется.*

1) Первый вид инициализации класса – это *автоматическая инициализация* полей классов в значения по умолчанию.

➤ Значение по умолчанию для типа **char** - это символ 'NUL', который означает, что это не отображаемый символ.

Блоки инициализации

Пример 1:

```
public class InitDemo1 {  
    private static char ch;  
    private static boolean bb;  
    private static byte by;  
    private static int ii;  
    private static float ff;  
    private static String str;  
    private static int[] array;  
    public static void main(String[] arg){  
        System.out.println("char: " + ch);  
        System.out.println("boolean: " + bb);  
        System.out.println("byte: " + by);  
        //...  
        System.out.println("String: " + str);  
        System.out.println("Array: " + array);  
    }  
}
```

Console output:

```
char:  
boolean: false  
byte: 0  
int: 0  
float: 0.0  
String: null  
Array: null
```

Блоки инициализации

2) Второй вид инициализации класса - это явные инициализаторы полей класса в их начальные значения (каждое поле класса может явно быть проинициализировано некоторым значением и инициализацию можно записать в одну строку).

```
public class InitDemo2 {  
    private static char ch = 'A';  
    private static boolean bb = true;  
    private static byte by = -56;  
    private static int ii = 1000;  
    private static float ff = 1.25e-2F;  
    private static String str = "Data";  
    private static int[] array = {0, 1, 2, 3};  
    // .....  
}
```

Блоки инициализации

- ❑ Компилятор Java автоматически генерирует метод инициализации класса (внутренний метод с именем **<clinit>**) для каждого класса.
- ✓ Метод гарантированно будет вызываться только один раз, когда класс впервые используется.
- ✓ Выражения инициализации полей класса вставляются в метод инициализации класса в порядке их появления в исходном коде (в выражении инициализации для поля класса можно использовать ранее объявленные поля класса).

```
public class InitDemo3 {
```

```
//...
```

```
private static byte by = 17;
```

```
private static int ii = 24 * by;
```

```
//...
```

```
}
```

Обратное не
допускается

Блоки инициализации

- ✓ В выражении инициализации поля класса можно использовать обращение к статическому методу (преимущество - повторное использование, если вам нужно инициализировать поле класса).

```
public class InitDemo4 {  
    private static int ii = initSt();  
    //...  
    private static int initSt() {  
        System.out.println("Init ii value");  
        return 1000;  
    }  
    //...  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        System.out.println("int: " + ii);  
    }  
}
```

Console output:

```
Init ii value  
Main  
int: 1000
```


Блоки инициализации

3) Третий вид инициализации – это статические блоки инициализации, используются когда требуется некоторая логика (например, обработка ошибок или циклы для заполнения сложных наборов данных).

Ограничения:

- оператор **return** не может использоваться в пределах статического блока инициализатора;
 - ключевое слово **this** не может использоваться в пределах статического блока инициализатора;
 - на не статическую переменную нельзя ссылаться из статического блока инициализатора.
- ❑ Компилятор Java вставляет код статического блока в метод инициализации класса (метод **<clinit>**) после инициализации полей класса выражением.

Блоки инициализации

Пример 2:

```
public class InitDemo5 {  
    private static char[] alph;  
  
    public static void main(String[] arg) {  
        System.out.print(Arrays.toString(alph));  
    }  
  
    static {  
        alph = new char[26];  
        int i = 0;  
        for (char c = 'a'; i < alph.length; c++, i++) {  
            alph[i] = c;  
        }  
    }  
}
```



Блоки инициализации

Особенности

- ✓ Класс может иметь любое количество статических блоков инициализации;
- ✓ Они могут появляться в любом месте тела класса;
- ✓ Исполнительная система гарантирует, что статические блоки инициализации вызываются в том порядке, в котором они появляются в исходном коде;
- ✓ Такой блок выполняется только один раз, когда класс инициализируется или загружается.

Блоки инициализации

- 4) Четвертый вид инициализации – не статические блоки инициализации, другими словами логические блоки, которые являются альтернативой конструкторам класса для инициализации полей экземпляра.

Выглядят :

```
{  
    // Любой код, необходимый для инициализации  
}
```

Используются для разделения блока кода между несколькими конструкторами.

Блоки инициализации

Пример 3:

```
public class Student {  
    private static int numOfStudents;  
    //...  
    public Student() {  
        //...  
        numOfStudents++;  
    }  
    public Student(String name) {  
        //...  
        numOfStudents++;  
    }  
}
```

Дублирование
кода

```
public class Student {  
    private static int numOfStudents;  
    //...  
    {  
        numOfStudents++;  
    }  
    public Student() {  
        //...  
    }  
    public Student(String name) {  
        //...  
    }  
}
```

Вынесение общего
кода в логический
блок

Блоки инициализации

Порядок инициализации класса

Инициализация полей класса значения на умолчанию

Инициализация полей класса выражениями

Выполнение статических блоков инициализации

Если это класс с точкой входа, то выполнение метода *main()*

Блоки инициализации

Порядок инициализации при создании экземпляра класса

Рекурсивный вызов и выполнение конструкторов суперклассов

Инициализация полей экземпляра значениями по умолчанию или начальными значениями

Выполнение логических блоков инициализации

Выполнение тела конструктора класса

Блоки инициализации

Пример 4:

```
public class InitDemo6 {  
    private int a = 5;  
    private static int b = 100;  
    { a = -5;  
      System.out.println("Logical block");  
    }  
    public InitDemo6() {  
        a = 10;  
        System.out.println("Constructor");  
    }  
    static { b = -5;  
        System.out.println("Static block");  
    }  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        InitDemo6 obj = new InitDemo6();  
        System.out.println("a=" + obj.a);  
    }  
}
```

Console output:

Static block

Main

Logical block

Constructor

a=10

Блоки инициализации

Инициализация переменной типа **final**

- ☐ должна быть инициализирована в той же строке, в которой и объявлена;
- ☐ должна быть инициализирована в каждом конструкторе;
- ☐ должна быть инициализирована в одном из логических блоков класса.

Потому что, переменная типа **final** может быть инициализирована только один раз.

Блоки инициализации

Пример 5:

```
public class InitDemo7 {  
    private final int xx = 50;  
    private final int zz;  
    private final int yy;  
    {  
        zz = 20;  
        System.out.println("Non-static block");  
    }  
    public InitDemo7() {  
        yy = 30;  
        System.out.println("Constructor");  
    }  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        InitDemo7 obj = new InitDemo7();  
    }  
}
```