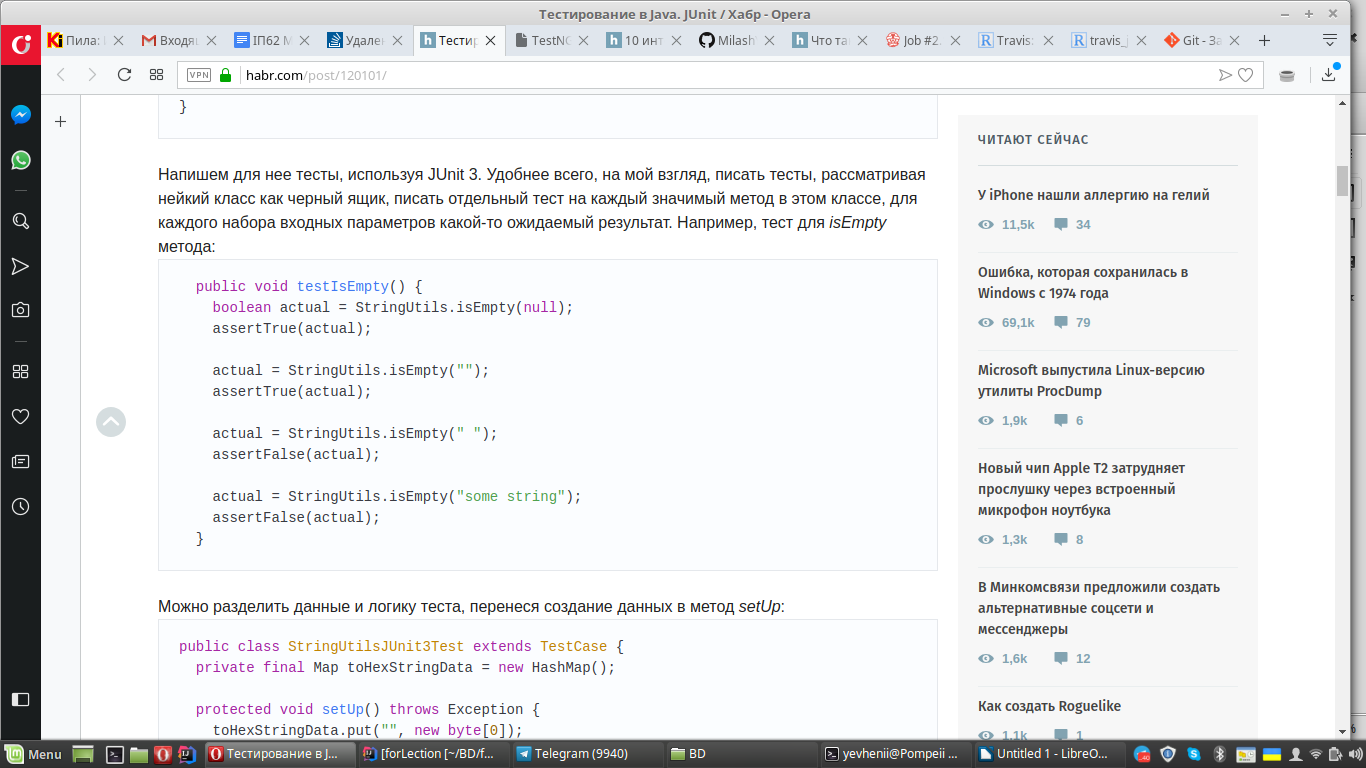
Сегодня все большую популярность приобретает [test-driven development(TDD)](http://ru.wikipedia.org/wiki/Разработка_через_тестирование), техника разработки ПО, при которой сначала пишется тест на определенный функционал, а затем пишется реализация этого функционала. На практике все, конечно же, не настолько идеально, но в результате код не только написан и протестирован, но тесты как бы неявно задают требования к функционалу, а также показывают пример использования этого функционала.  
  
Итак, техника довольно понятна, но встает вопрос, что использовать для написания этих самых тестов?   
  
Ну и начну с, пожалуй, самого известного, а потому и самого используемого фреймворка для тестирования — [JUnit](http://junit.org/). Используется он в трёх вариантах JUnit 3 и JUnit 4 и JUnit 5. В старых проектах до сих пор используется 3-я, которая поддерживает Java 1.4.

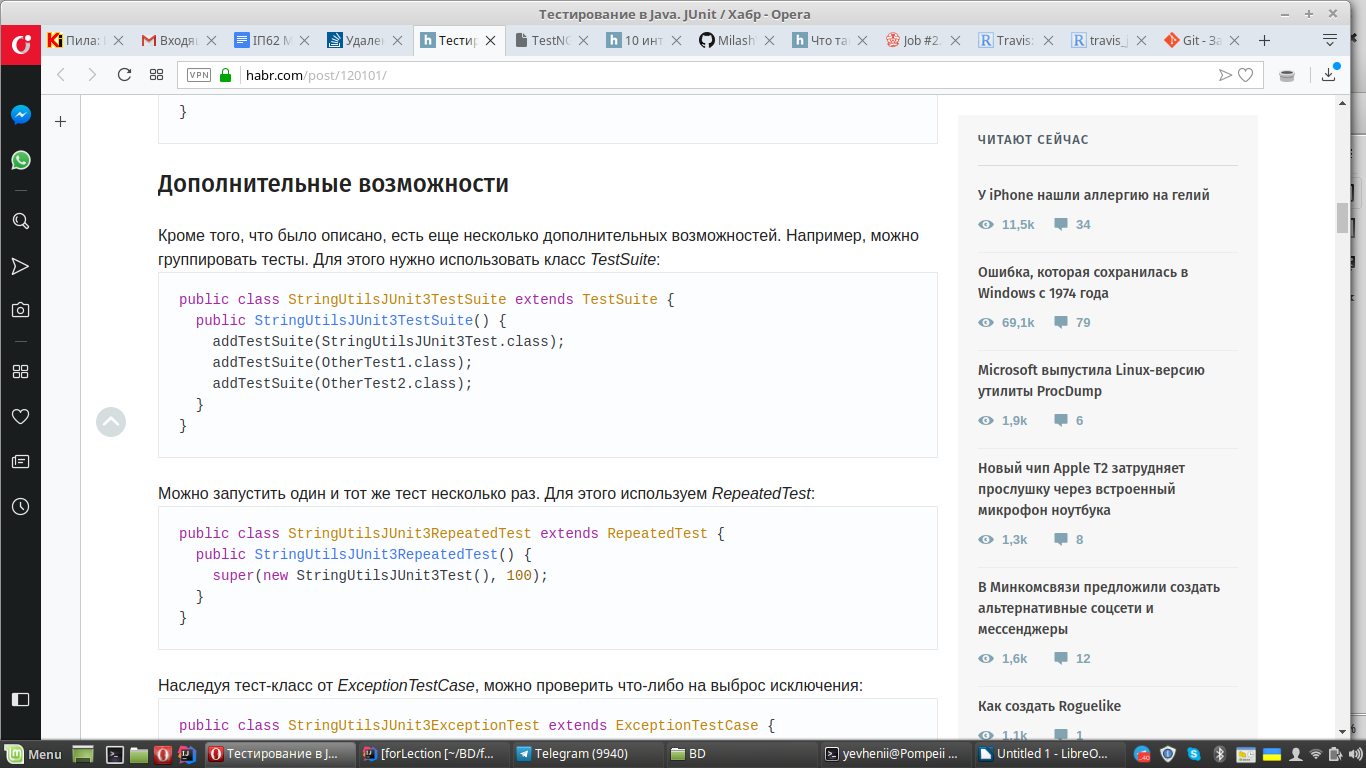
# JUnit 3

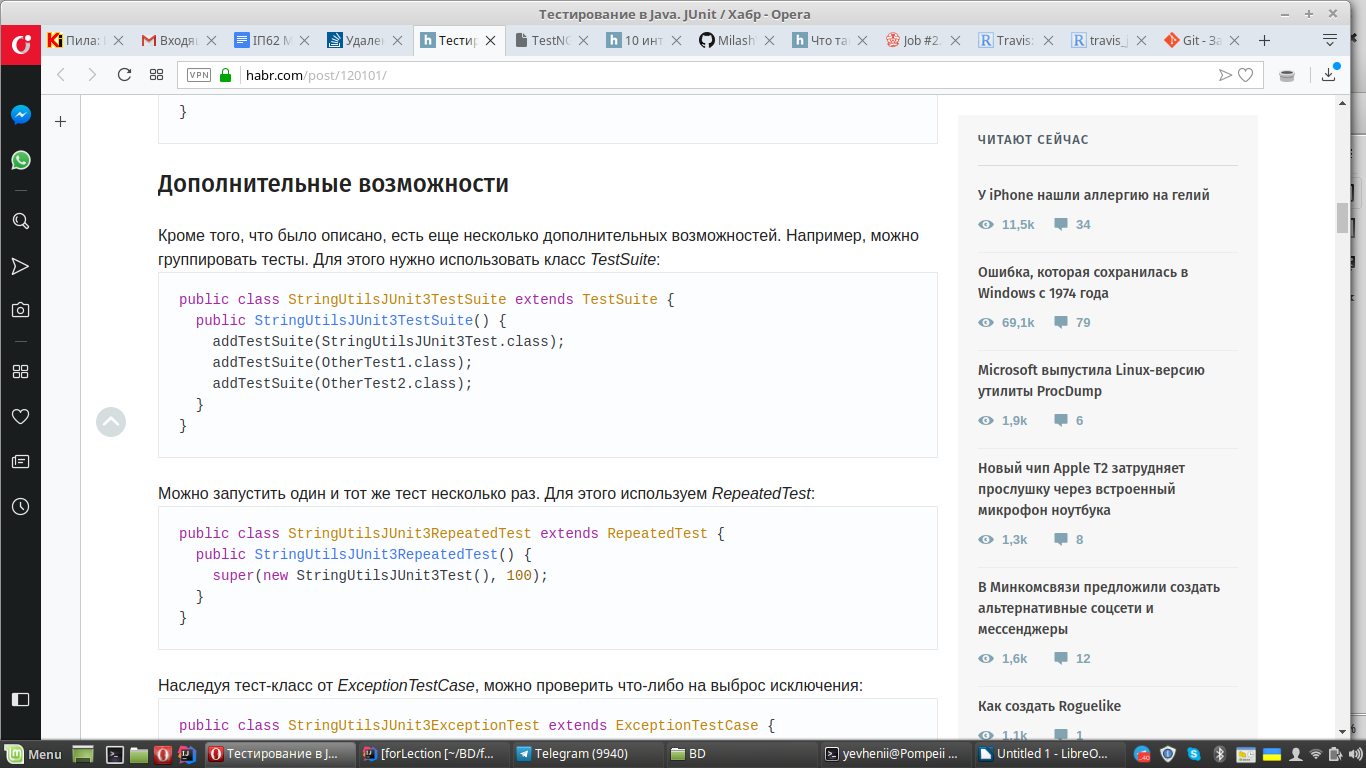
Для создания теста нужно унаследовать тест-класс от *TestCase*, переопределить методы *setUp* и *tearDown* если надо, ну и самое главное — создать тестовые методы(должны начинаться с *test*). При запуске теста сначала создается экземляр тест-класса(для каждого теста в классе отдельный экземпляр класса), затем выполняется метод *setUp*, запускается сам тест, ну и в завершение выполняется метод *tearDown*. Если какой-либо из методов выбрасывает исключение, тест считается провалившимся.  
  
Примечание: тестовые методы должны быть *public void*, могут быть *static*.  
  
Сами тесты состоят из выполнения некоторого кода и проверок. Проверки чаще всего выполняются с помощью класса *Assert* хотя иногда используют ключевое слово *assert*.

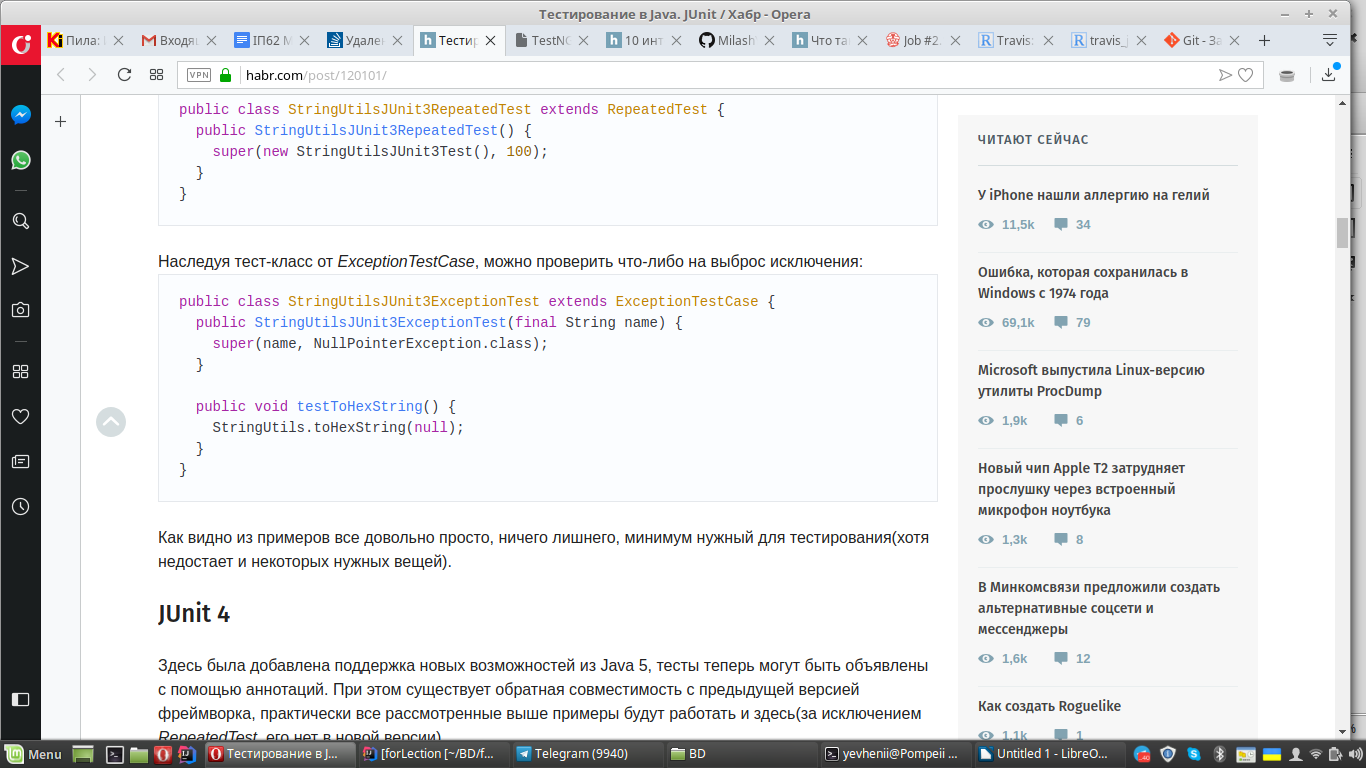


## **Дополнительные возможности**

Кроме того, что было описано, есть еще несколько дополнительных возможностей. Например, можно группировать тесты. Для этого нужно использовать класс *TestSuite*.

Можно запустить один и тот же тест несколько раз. Для этого используем *RepeatedTest*.

Наследуя тест-класс от *ExceptionTestCase*, можно проверить что-либо на выброс исключения.

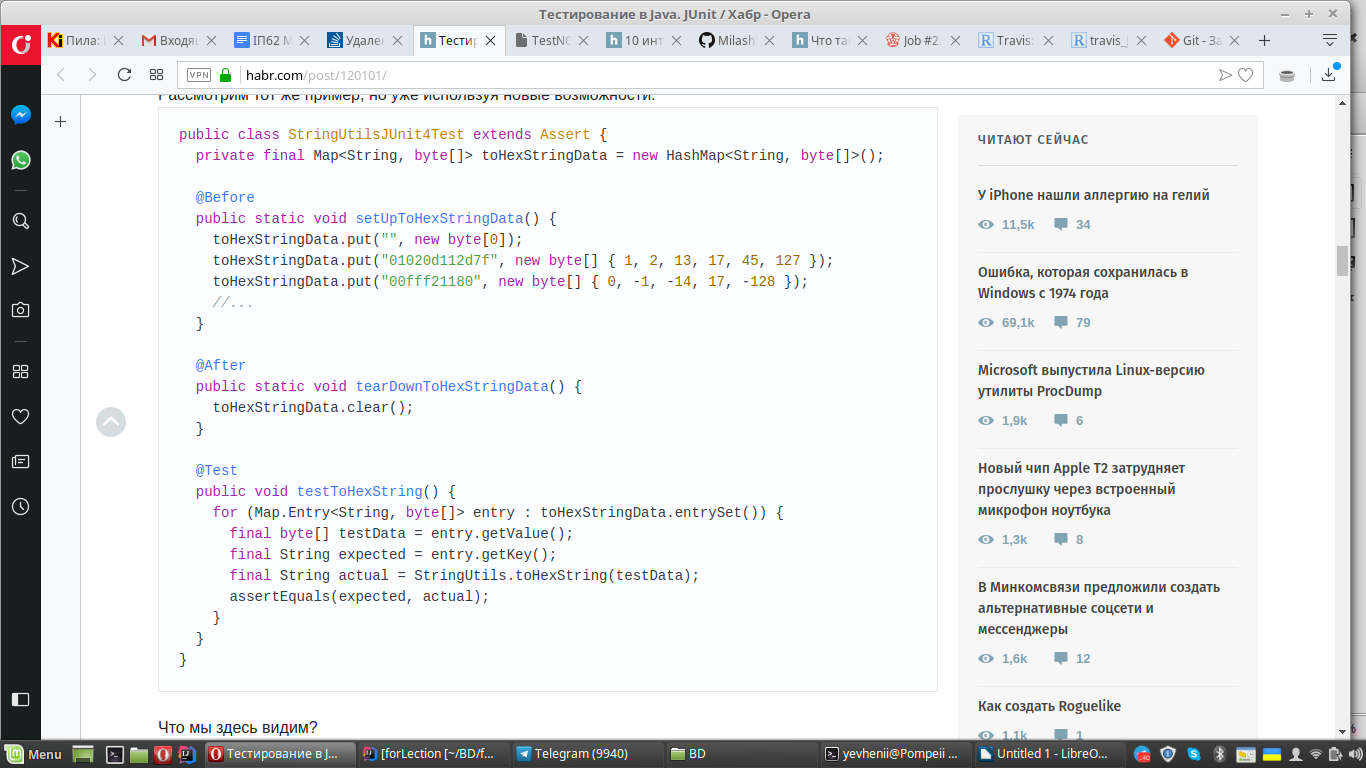


**JUnit 4**

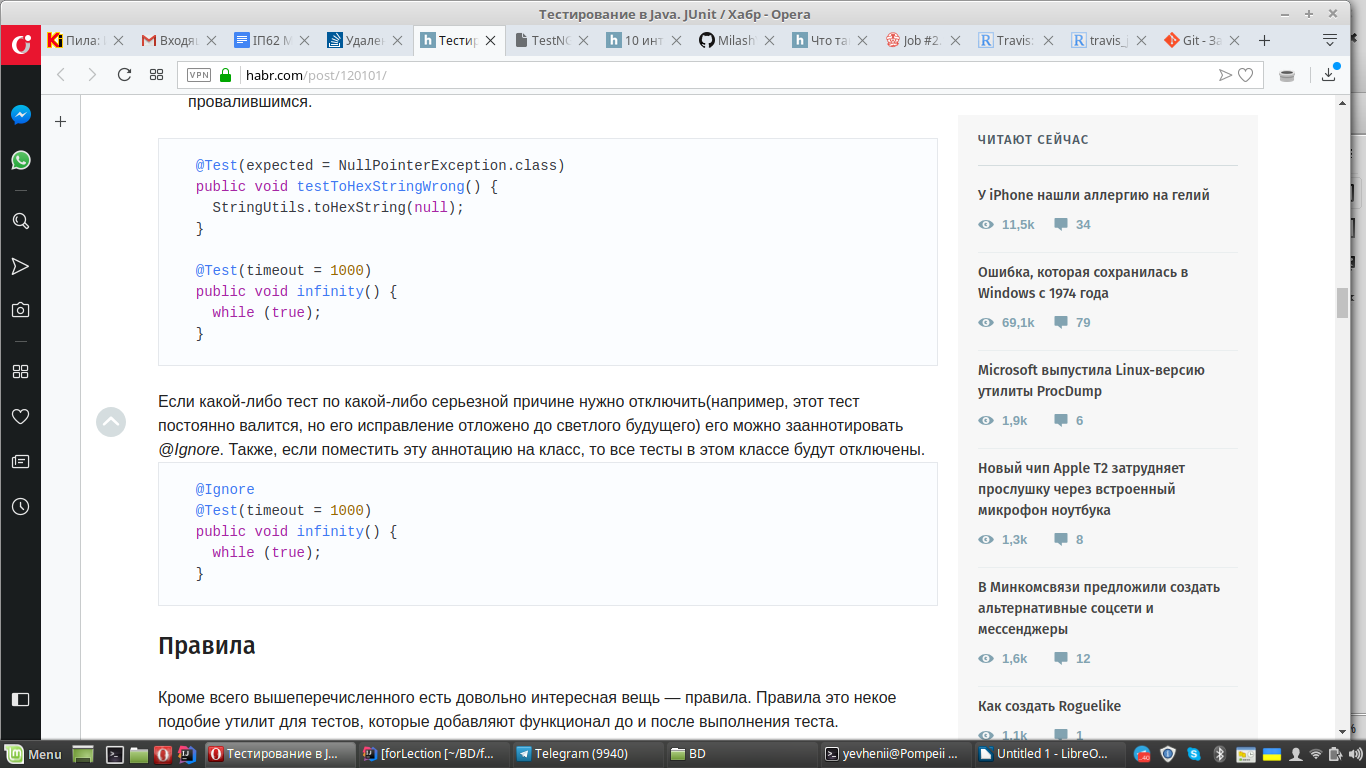
Здесь была добавлена поддержка новых возможностей из Java 5, тесты теперь могут быть объявлены с помощью аннотаций. При этом существует обратная совместимость с предыдущей версией фреймворка, практически все рассмотренные выше примеры будут работать и здесь(за исключением *RepeatedTest*, его нет в новой версии).  
  
Итак, что же поменялось?

## Основные аннотации

* Аннотация [*Before*](https://habrahabr.ru/users/before/) обозначает методы, которые будут вызваны до исполнения теста, методы должны быть *public void*. Здесь обычно размещаются предустановки для теста, в нашем случае это генерация тестовых данных (метод *setUpToHexStringData*).
* Аннотация *@BeforeClass* обозначает методы, которые будут вызваны до создания экземпляра тест-класса, методы должны быть *public static void*. Имеет смысл размещать предустановки для теста в случае, когда класс содержит несколько тестов использующих различные предустановки, либо когда несколько тестов используют одни и те же данные, чтобы не тратить время на их создание для каждого теста.
* Аннотация [*After*](https://habrahabr.ru/users/after/) обозначает методы, которые будут вызваны после выполнения теста, методы должны быть *public void*. Здесь размещаются операции освобождения ресурсов после теста, в нашем случае — очистка тестовых данных (метод *tearDownToHexStringData*).
* Аннотация *@AfterClass* связана по смыслу с *@BeforeClass*, но выполняет методы после теста, как и в случае с *@BeforeClass*, методы должны быть *public static void*.
* Аннотация [*Test*](https://habrahabr.ru/users/test/) обозначает тестовые методы. Как и ранее, эти методы должны быть *public void*. Здесь размещаются сами проверки. Кроме того, у данной аннотации есть два параметра, *expected*— задает ожидаемое исключение и *timeout* — задает время, по истечению которого тест считается провалившимся.

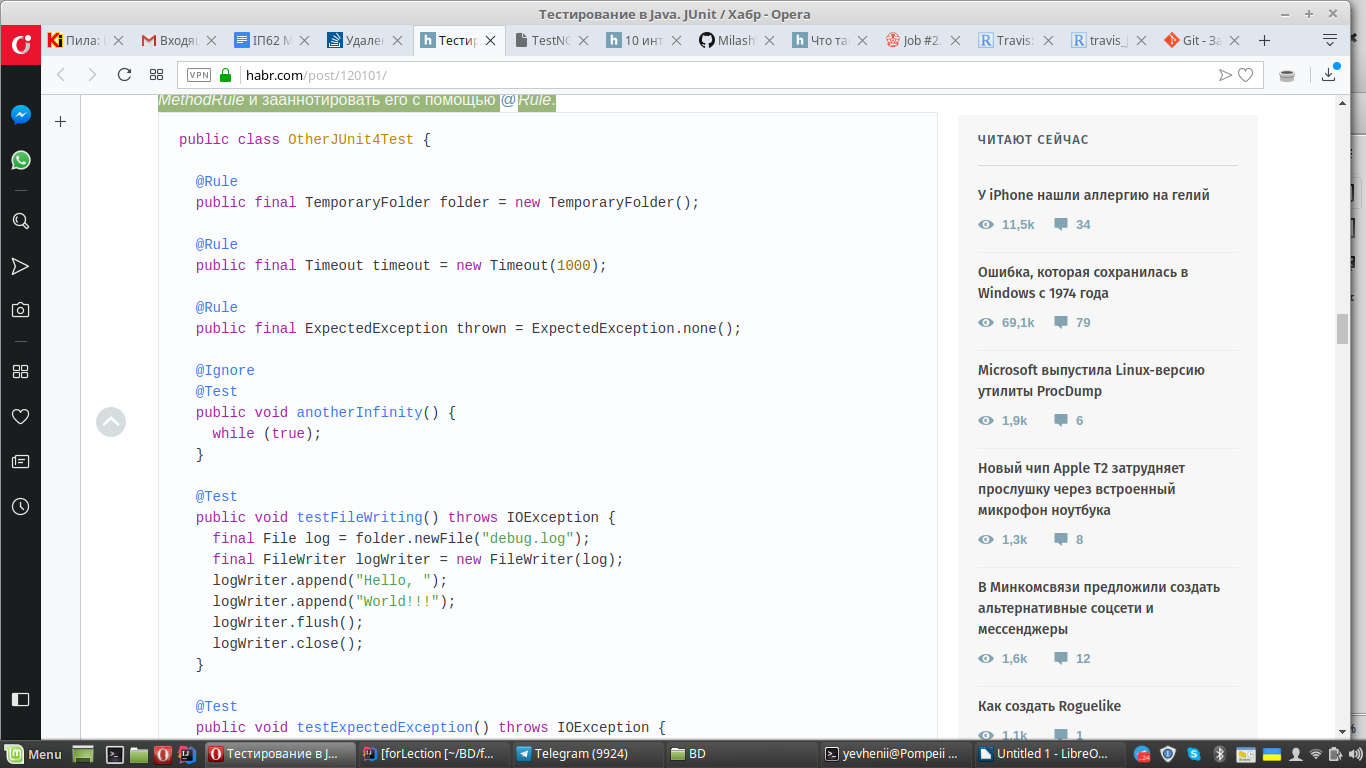


Если какой-либо тест по какой-либо серьезной причине нужно отключить(например, этот тест постоянно валится, но его исправление отложено до светлого будущего) его можно зааннотировать *@Ignore*. Также, если поместить эту аннотацию на класс, то все тесты в этом классе будут отключены.



## **Правила**

Кроме всего вышеперечисленного есть довольно интересная вещь — правила. Правила это некое подобие утилит для тестов, которые добавляют функционал до и после выполнения теста.  
  
Например, есть встроенные правила для задания таймаута для теста(*Timeout*), для задания ожидаемых исключений(*ExpectedException*), для работы с временными файлами(*TemporaryFolder*) и д.р. Для объявления правила необходимо создать *public* не *static* поле типа производного от *MethodRule* и зааннотировать его с помощью [*Rule*](https://habrahabr.ru/users/rule/).

****

**JUnit 5**

Итак, официальный сайт начинает с того, что сообщает нам о новом строении JUnit:

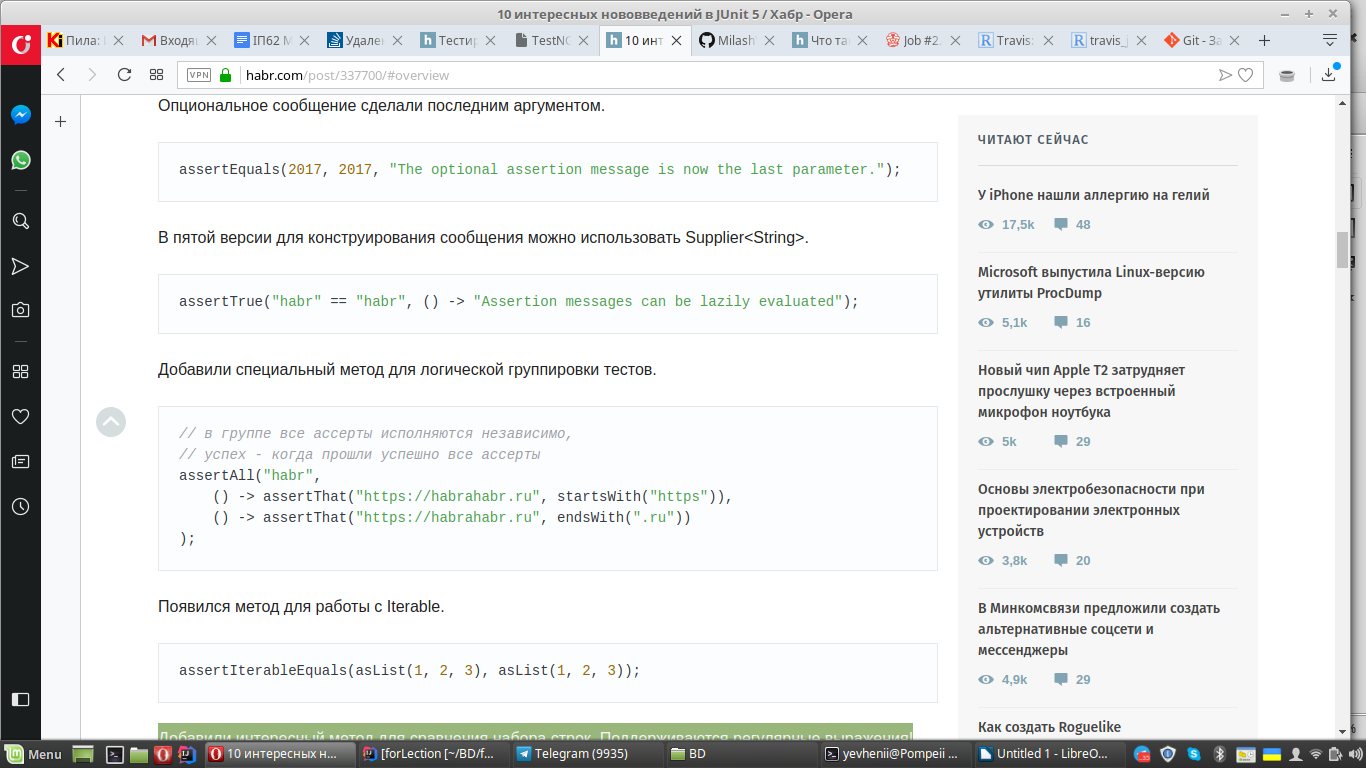
JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage .

***JUnit Platform*** — фундаментальная основа для запуска на JVM фреймворков для тестирования. Платформа предоставляет [TestEngine API](http://junit.org/junit5/docs/current/api/org/junit/platform/engine/TestEngine.html), для разработки фреймворков (для тестирования), которые могут быть запущены на платформе. Кроме этого, в платформе имеется [Console Launcher](http://junit.org/junit5/docs/current/user-guide/" \l "running-tests-console-launcher) для запуска платформы из коммандной строки а также для запуска любого JUnit 4 Runner'а на платформе. Уже, кстати, есть плагины для [Gradle](http://junit.org/junit5/docs/current/user-guide/" \l "running-tests-build-gradle) и [Maven](http://junit.org/junit5/docs/current/user-guide/" \l "running-tests-build-maven).  
  
***JUnit Jupiter*** — сердце JUnit 5. Этот проект предоставляет новые возможности для написания тестов и создания собственных расширений. В проекте реализован специальный TestEngine для запуска тестов на ранее описанной платформе.  
  
***JUnit Vintage*** — поддержка легаси. Определяется TestEngine для запуска тестов ориентированных на JUnit 3 и JUnit 4.

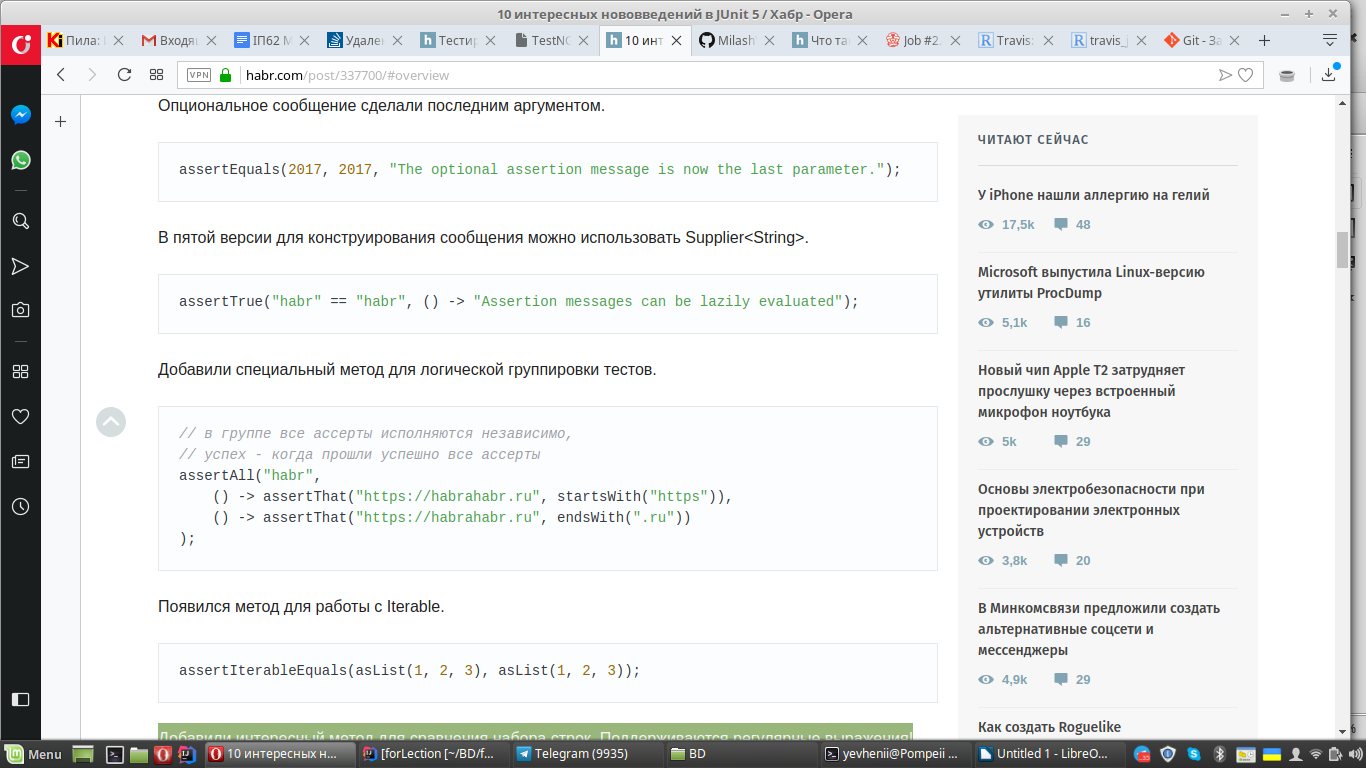
#### **Обзор нововведений**

А теперь наконец-то перейдем к примерам!  
  
 **public — всё**  
JUnit больше не требует, чтобы методы были публичными.

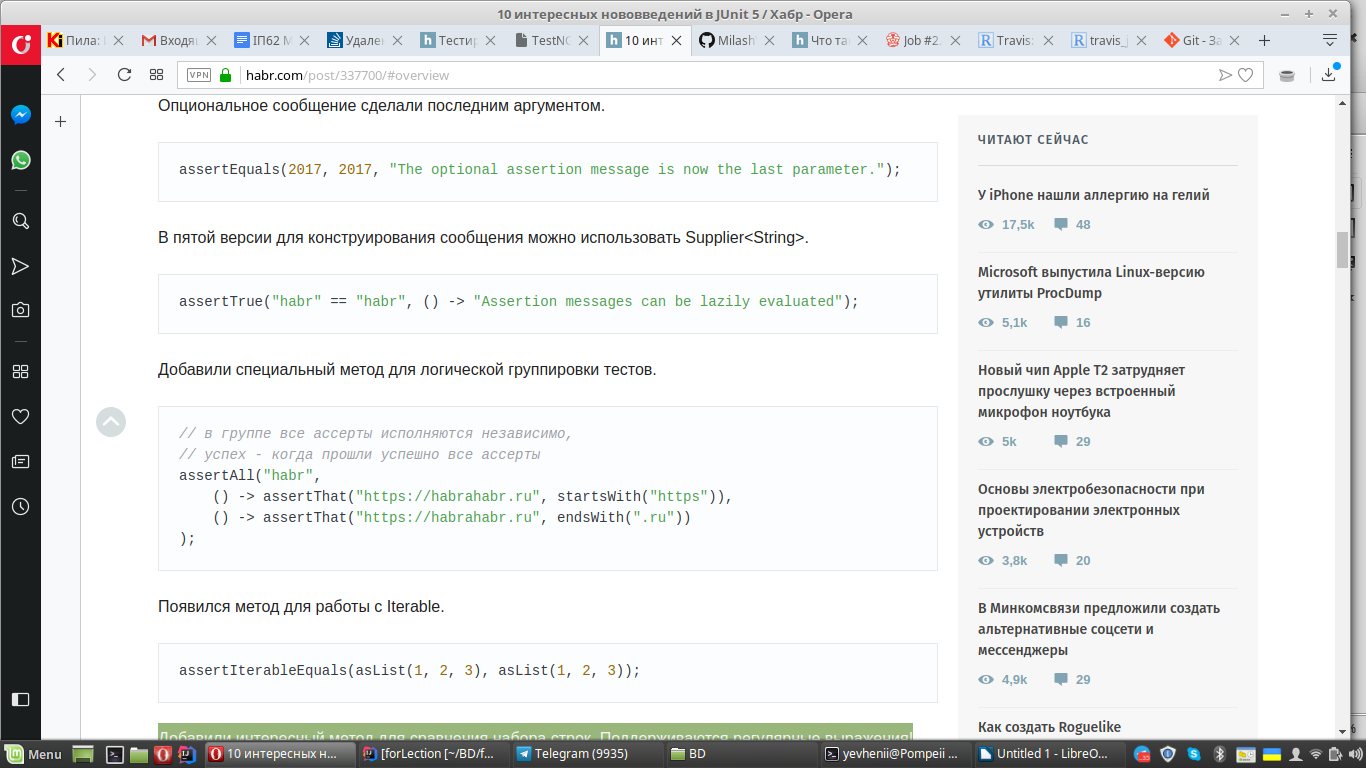
**Продвинутый assert**  
Опциональное сообщение сделали последним аргументом.



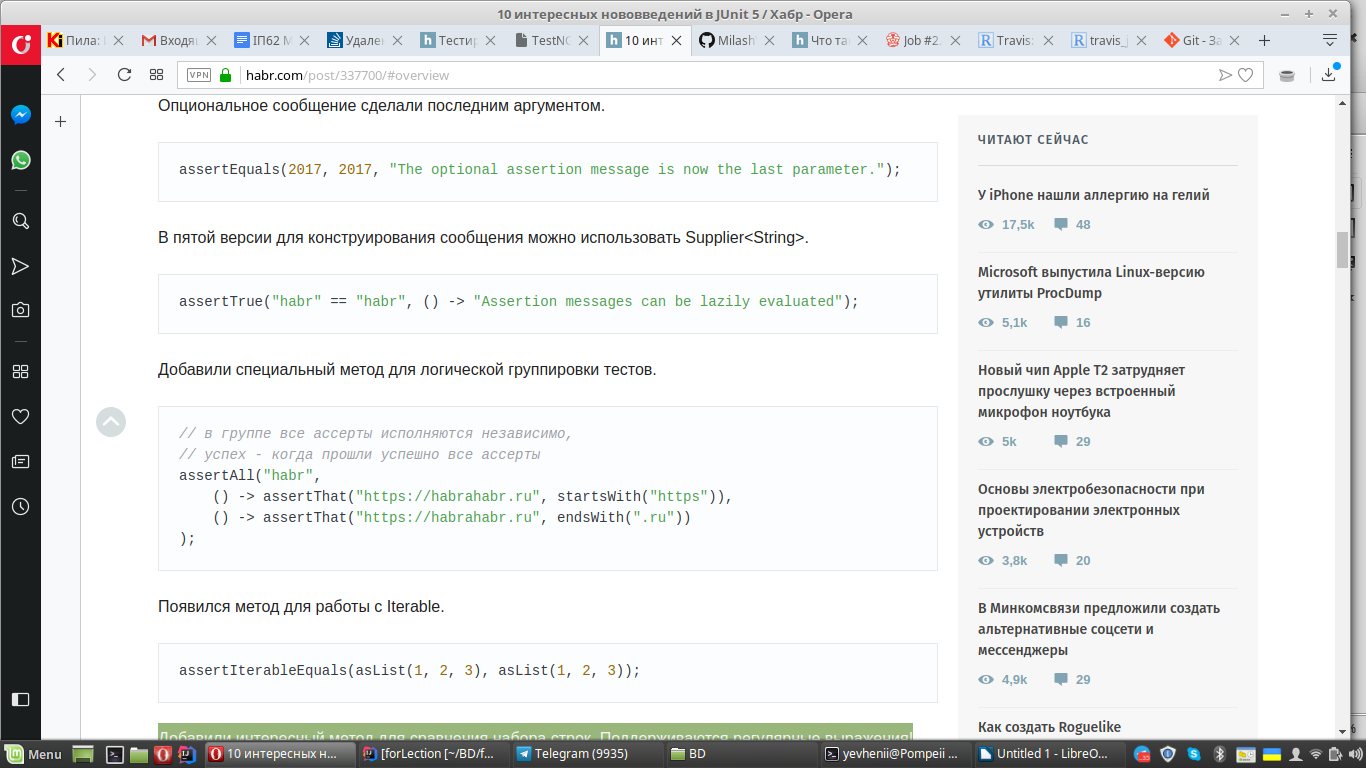
В пятой версии для конструирования сообщения можно использовать Supplier<String>.



Добавили специальный метод для логической группировки тестов.

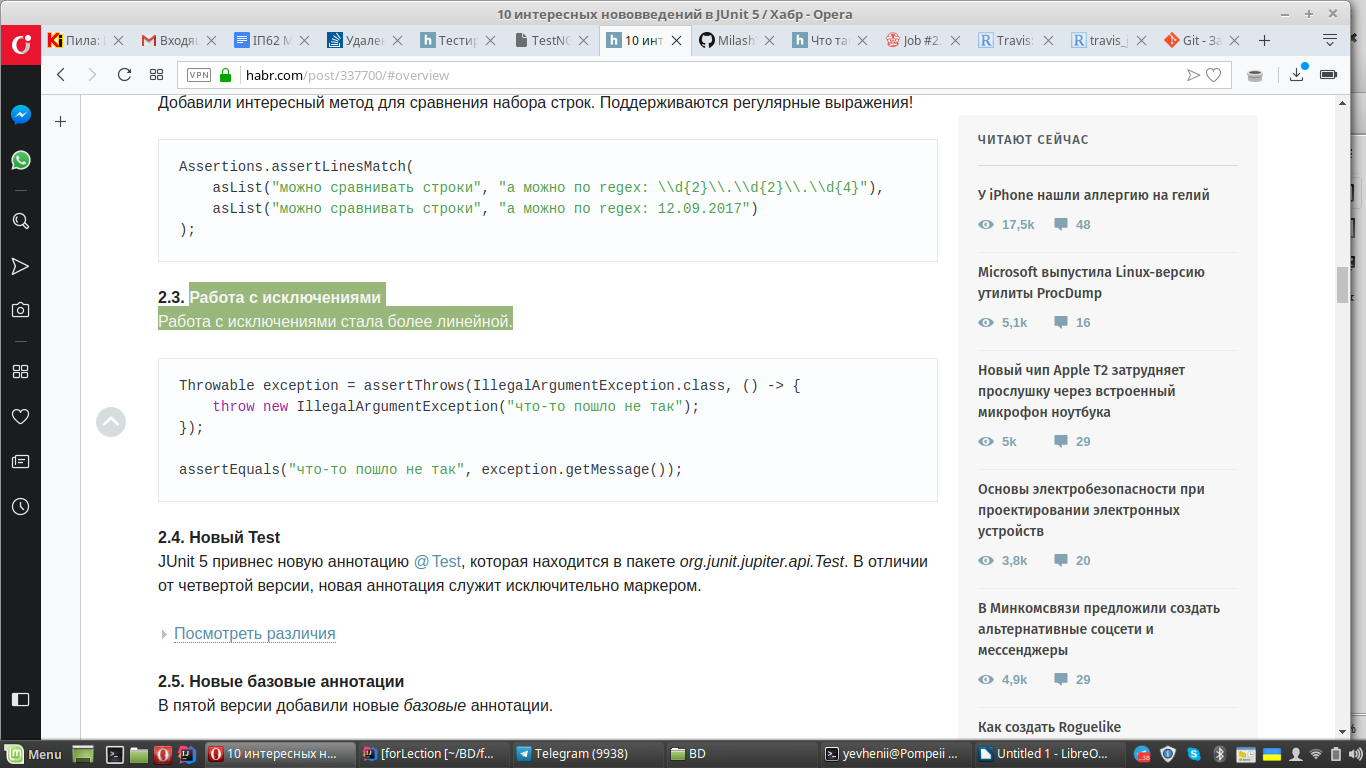


Появился метод для работы с Iterable.



Добавили интересный метод для сравнения набора строк. Поддерживаются регулярные выражения.

**Работа с исключениями**  
Работа с исключениями стала более линейной.

**Новый Test**

JUnit 5 привнес новую аннотацию [Test](https://habrahabr.ru/users/test/), которая находится в пакете *org.junit.jupiter.api.Test*. В отличии от четвертой версии, новая аннотация служит исключительно маркером.

**Вложенные классы**  
Аннотация @Nested позволяет использовать внутренние классы при разработке тестов, что позволяет иногда более удобным способом группировать/дополнять тесты.

**Разделяемый инстанс класса для запуска тестов**  
Для гарантии независимости и изоляциии тестов JUnit во всех предыдущих версиях всегда создавал по инстансу на тест (т.е. на каждый запуск метода отдельный инстанс). В пятой версии такое поведение можно изменить используя новую аннотацию @TestInstance(Lifecycle.PER\_CLASS). В таком случае инстанс будет создан только один раз и будет переиспользован для запуска всех тестов, определенных внутри этого класса.

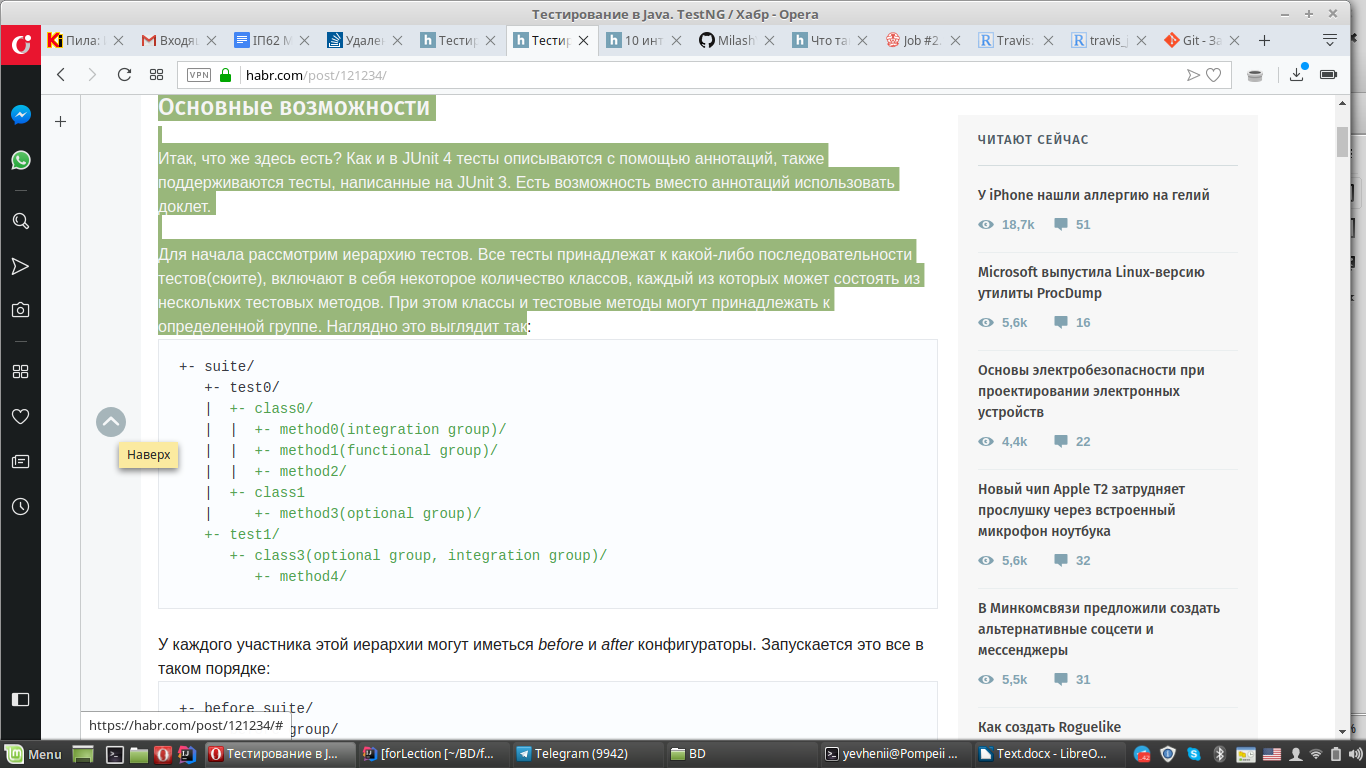
**Автоматический повторный запуск теста**  
Еще одна приятная добавка! Аннотация @RepeatedTest сообщает JUnit, что данный тест нужно запустить несколько раз. При этом, каждый такой вызов будет независимым тестом, а значит для него будут работать аннотации @BeforeAll, @BeforeEach, @AfterEach и @AfterAll.

**Параметризированные тесты**  
Параметризированные тесты позволяют запускать тест несколько раз с различными входными данными. На данный момент поддерживаются только данные примитивных типов: *int*, *long*, *double*, *String*. Но не стоит отчаиваться! JUnit 5 определяет несколько дополнительных аннотаций для указания источника данных для параметризированных тестов. Итак, начнём!

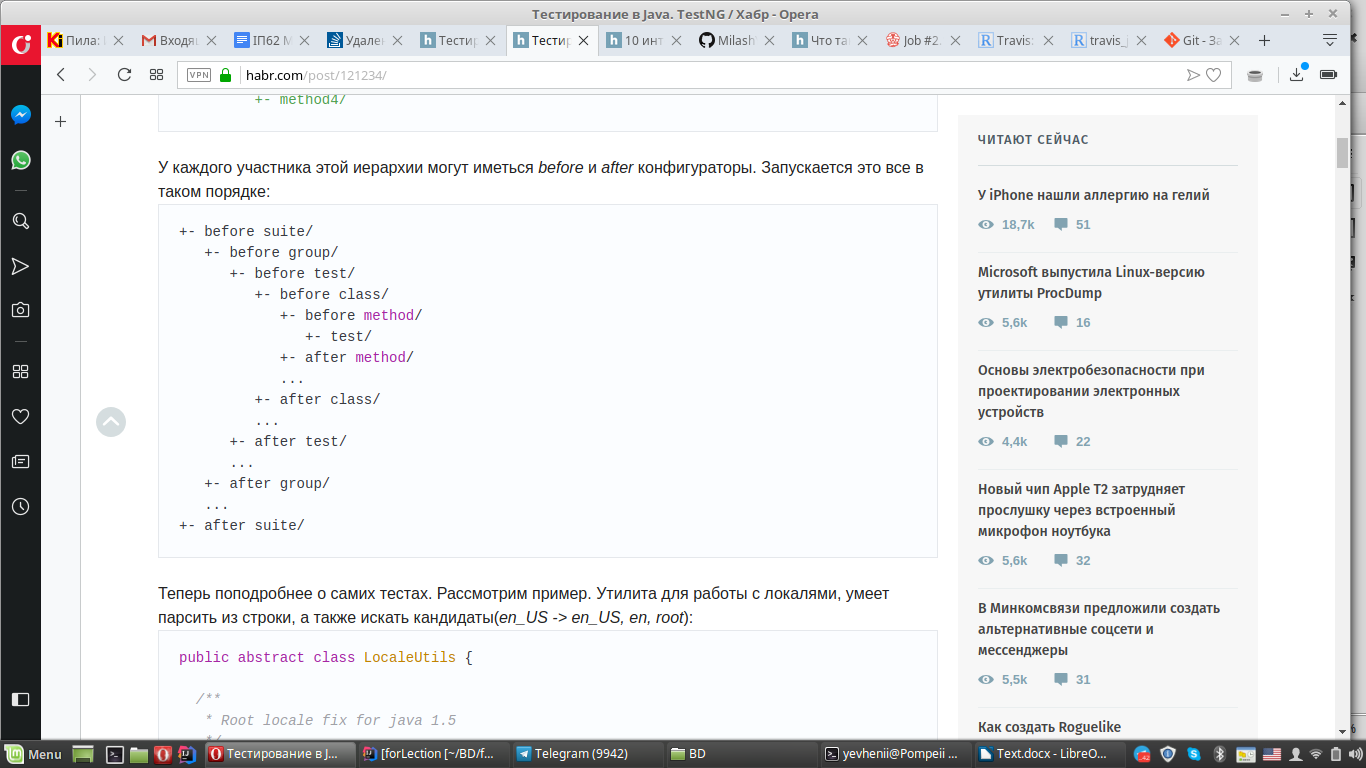
Наверняка все знакомы с таким понятием как [test-driven development(TDD)](http://ru.wikipedia.org/wiki/Разработка_через_тестирование). Наряду с ним также существует такое понятие, как [data-driven testing(DDT)](http://en.wikipedia.org/wiki/Data-driven_testing)— техника написания тестов, при которой данные для тестов хранятся отдельно от самих тестов. Они могут храниться в базе данных, файле, генерироваться во время исполнения теста. Это очень удобно, так как один и тот же функционал тестируется на различных наборах данных, при этом добавление, удаление или изменение этих данных максимально упрощено.

# Основные возможности

Итак, что же здесь есть? Как и в JUnit 4 тесты описываются с помощью аннотаций, также поддерживаются тесты, написанные на JUnit 3. Есть возможность вместо аннотаций использовать доклет.  
  
Для начала рассмотрим иерархию тестов. Все тесты принадлежат к какой-либо последовательности тестов(сюите), включают в себя некоторое количество классов, каждый из которых может состоять из нескольких тестовых методов. При этом классы и тестовые методы могут принадлежать к определенной группе. Наглядно это выглядит так:



У каждого участника этой иерархии могут иметься *before* и *after* конфигураторы. Запускается это все в таком порядке:



**Тут може бути ваш код))**

Что здесь есть?

* Как уже было сказано я предпочитаю наследовать тест-класс от *Assert*, это можно заменить статическим импортом, либо использованием класса напрямую(*Assert.assertEquals(...)*). В реальной системе удобнее всего наследовать тест от какого-либо базового класса, который в свою очередь наследовать от *Assert*, это дает возможность переопределять либо добавлять необходимые методы. Внимание: в отличие от такого же класса в JUnit здесь во все методы актуальное значение передается первым, ожидаемое вторым(в JUnit наоборот).
* Аннотации *@BeforeSuite*, *@AfterSuite* обозначают методы, которые исполняются единожды до/после исполнения всех тестов. Здесь удобно располагать какие-либо тяжелые настройки общие для всех тестов, например, здесь можно создать пул соединений с базой данных.
* Аннотации *@BeforeTest*, *@AfterTest* обозначают методы, которые исполняются единожды до/после исполнения теста(тот, который включает в себя тестовые классы, не путать с тестовыми методами). Здесь можно хранить настройки какой-либо группы взаимосвязанных сервисов, либо одного сервиса, если он тестируется несколькими тест-классами.
* Аннотации *@BeforeClass*, *@AfterClass* обозначают методы, которые исполняются единожды до/после исполнения всех тестов в классе, идентичны предыдущим, но применимы к тест-классам. Наиболее применим для тестирования какого-то определенного сервиса, который не меняет свое состояние в результате теста.
* Аннотации *@BeforeMethod*, *@AfterMethod* обозначают методы, которые исполняются каждый раз до/после исполнения тестового метода. Здесь удобно хранить настройки для определенного бина или сервиса, если он не меняет свое состояние в результате теста.
* Аннотации *@BeforeGroups*, *@AfterGroups* обозначает методы, которые исполняются до/после первого/последнего теста принадлежащего к заданным группам.
* Аннотация *@Test* обозначает сами тесты. Здесь размещаются проверки. Также применима к классам

У всех этих аннотаций есть следующие параметры:

* enabled — можно временно отключить, установив значение в false
* groups — обозначает, для каких групп будет исполнен
* inheritGroups — если true(а по умолчанию именно так), метод будет наследовать группы от тест-класса
* timeOut — время, после которого метод «свалится» и потянет за собой все зависимые от него тесты
* description — название, используемое в отчете
* dependsOnMethods — методы, от которых зависит, сначала будут выполнены они, а затем данный метод
* dependsOnGroups — группы, от которых зависит
* alwaysRun — если установить в true, будет вызываться всегда независимо от того, к каким группам принадлежит, не применим к *@BeforeGroups*, *@AfterGroups*

Как видно из примера тест практически ничем не отличается от такого же теста на JUnit. Если нет разницы, то зачем использовать TestNG?

**Тут може бути важ код))**

Проще? Конечно, данные хранятся отдельно от самого теста. Удобно? Конечно, можно добавлять тесты, добавляя всего лишь строчку в метод parseLocaleData.Итак, как это работает?

* Объявляем тестовый метод со всеми нужными ему параметрами, например входные и ожидаемые данные. В нашем случае это строка, которую нужно распарсить в локаль и ожидаемая в результате локаль.
* Объявляем дата провайдер, хранилище данных для теста. Обычно это метод, возвращающий *Object[][]* либо *Iterator<Object[]>*, содержащий список параметров для определенного теста, например *{«en\_US», Locale.US}*. Этот метод должен быть зааннотирован с помощью *@DataProvider*, в самом тесте он объявляется с помощью параметра *dataProvider* в аннотации *@Test*. Также можно указать имя(параметр *name*), если не указывать в качестве имени будет использоваться название метода.