

# **Introduction to Software Testing (2nd edition) Chapter 3**

## **Test Automation**

Paul Ammann & Jeff Offutt

<http://www.cs.gmu.edu/~offutt/softwaretest/>

*Updated October 2018*

---

# What is Test Automation?

The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions

- Reduces cost
- Reduces human error
- Significantly reduces the cost of regression testing

test automation یا خودکار سازی تست:

از همون ابتدا میتوانیم اتومات کردن تست ها رو انجام بدیم

ایا همیشه میشه 100 درصد خودکار سازی رو داشته باشیم؟ نه یکسری از ویژگی ها هستن که ما به

عنوان ورودی نمی تونیم بدیم و نیاز به شبیه سازی داره ینی نه تنها این که از اون کدهایی که دارن

خودکار سازی تست رو انجام میدن استفاده کنم بلکه باید یک نرم افزار های دیگه هم استفاده بکنیم یا

خودمون بیایم دستی اون محیط رو شبیه سازی بکنیم تا اینکه نرم افزارمون رو تست بکنیم برای

ورودی هایی که کنترل کردن شون راحت نیست باید از این روش استفاده بکنیم

تعريف خودکار سازی تست: ینی از یک نرم افزار استفاده بکنیم برای اینکه اجرای تست رو کنترل

بکنه ینی اون نرم افزار به جای انسان بیاد اجرای تست رو کنترل بکنه و حواسش باشه اون معیار

های پوششی که مد نظر ما بوده قبول میشه یا نه و پر فرمنس خوبی رو هم برای این کار برای ما

داشته باشه --> مزیت: خطای انسانی کاهش پیدا کنه - هزینه ها کمتر بشه

# Software Testability (3.1)

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met

- Plainly speaking – how hard it is to find faults in the software
- Testability is dominated by two practical problems
  - How to provide the test values to the software
  - How to observe the results of test execution

یکی از نان فانکشنال ریکوارمنت ها **testability** است  
بنی این که تا چقدر خوب می توانیم با تست **fault** ها رو پیدا کنیم

# **Observability and Controllability**

## □ Observability

**How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components**

- Software that affects hardware devices, databases, or remote files have low observability

## □ Controllability

**How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors**

- Easy to control software with inputs from keyboards
- Inputs from hardware sensors or distributed software is harder

-  
دو تا نکته داره: testability

observability: ینی این که بتونیم خروجی هاش رو و تاثیراتی که روی محیط و یا سخت افزارها داره رو چقدر خوب مشاهده کنیم --> ینی رفتارها و output های نرم افزار رو تا چقدر خوب می تونیم بررسی بکنیم و تاثیراتی نرم افزار روی محیطش و سخت افزارها و نرم افزارهای دیگه داره چقدر controllability: کنترل کردن یک نرم افزار ینی این که چجوری بهش input بدیم و هر چقدر ساده تر بتونیم ورودی ها رو در اختیار یک نرم افزار قرار بدم ینی کنترل کردنش راحت تره و در نتیجه تست کردنش هم راحت تره --> پس این مربوط به ورودی ها هستش و تاثیرات محیط روی نرم افزار که تا چقدر خوب می تونیم مشاهده بکنیم و کنترل بکنیم

نکته: هر چقدر controllability و observability بهتر باشه testability هم بهتره  
مثلا اگر ورودی رو بتونیم از طریق صفحه کلید بدیم ینی یک نرم افزاری داریم که ورودی رو فقط از طریق صفحه کلید لازمه که دریافت کنه پس اینجا testability نرم افزار بالاست چون controllability اش در حد بالایی است

نکته: توی سیستم های توزیع شده هم observability و controllability پایین هستش پس در کل testability پایینه

مثال:

مثلاً اومدیم نرم افزار میکروکنترلر مربوط به هدایت هوایما رو نوشتم --> این یکی از شرایطی که باید هندل بکنه این هستش که اگر هوایما داره سقوط میکنه بره روی حالت اتوپاپیلوت --> نرم افزار نیاز به یک شرایطی داره که بتونه به این مرحله از تست برسه ینی توی شرایط سقوط قرار بگیره و بفهمه که هوایما واقعاً داره سقوط پیدا می کنه تا بتونه خروجیشو تولید بکنه و بره روی حالت اتوپاپیلوت

سقوط کردن هوایما نیاز به یک محیط شبیه سازی داره --> ینی یک تیم رو استخدام بکنیم و بهش بگیم برآمون یک نرم افزار شبیه سازی هوایما بنویس حالا اینایی که بالا گفته controllability میشه

: observability و برای

مثلاً نرم افزار داره توی یک شرایطی بازوی مکانیکی رو حرکت میده --> مثلاً توی شرایط اتش سوزی نرم افزار قراره بره به عنوان یک رباتی که اتش نشان است ??????

---

# **Components of a Test Case (3.2)**

- A test case is a multipart artifact with a definite structure
- Test case values

The input values needed to complete an execution of the software under test

- Expected results

The result that will be produced by the test if the software behaves as expected

- A *test oracle* uses expected results to decide whether a test passed or failed

test oracle : به اون برنامه نویس میگن و الزاما قرار نیست ک ادم باشه به نرم افزار هم گفته میشه --> هر انسان یا پروگرمی که بتونه نتایج مورد انتظار ما رو با output واقعی نرم افزار رو مقایسه بکنه و بهمون بگه تست پاس شده یا نشده میشه test oracle تست کیس: ینی توی تستمون چه ورودی هایی و چه خروجی هایی داریم --> ورودی ها چیان؟

## 1- Test case values و 2- Prefix values و 3- Postfix values

test case value ینی واقعا اون موردی که میخوایم تست بکنیم رو value براش فراهم بکنیم --> اینا هم همون مقداری بودن که بخاطر خود تست به برنامه می دادیم مثلا اگر بخوایم یک فانکشن رو تست بکنیم ورودی های اونتابع میشن input value ها ولی از وقتی مین کار خودشو شروع میکنه تا بررسیم به اون فانکشن خاص اگر باز به input value نیاز داشتیم اینا میشن prefix value ها چون ما میرسوند به لحظه ای فرآخوانی متدى که می خوایم تستش بکنیم و بعد از اینکه از postfix value باز ورودی نیاز داشته که برنامه خاتمه پیدا بکنه این میشه

# Affecting Controllability and Observability

## □ Prefix values

Inputs necessary to put the software into the appropriate state to receive the test case values

## □ Postfix values

Any inputs that need to be sent to the software after the test case values are sent

1. *Verification Values* : Values needed to see the results of the test case values
2. *Exit Values* : Values or commands needed to terminate the program or otherwise return it to a stable state

**Prefix values** : در حالت کلی اونایی میشه که مورد استفاده قرار میگیرن نه با خاطر تست بلکه با خاطر این که برنامه رو برسونن به اون استیتی که می خوایم تست بکنیم مثلا برنامه نویسی سیم کارت رو انجام دادیم و می خوایم زنگ بزنیم به یک نفر و ببینیم کیفیت تماس خوبه یا نه و ایا توى تماس، زمان تماس می افته یا نه --> برای اینکه ما بخوایم این کارو انجام بدیم به **values** احتیاج داریم یعنی یکسری مقدار هستن که ربطی به تست ما ندارن یعنی ربطی به این ندارن که الان توی تست سیم کارت میخوایم تماس رو برقرار کنیم پس ربط به چی دارن؟ فقط کمک میکن که نرم افزارمون از نقطه شروع به اون مرحله مورد نظر برسه پس اینا پیش نیازها هستن ولی خودشون نقشی توی تست ندارن

**Postfix values** : در حالت کلی اونایی هستن که می خوان برنامه رو بعد از تست برسونن به اون نقطه پایانی یا به جایی که می خود خاتمه پیدا بکنه کار ما بعد از **test case value** اینو داریم نرم افزار باید یکسری مراحل رو طی بکنه تا به نقطه **exit** اش برسه ولی اون **value**ها هنوز برای ما مهم نیستن که بهش میگن **postfix value** یعنی مقادیری که پسوندی هستن یعنی تستمون انجام شده و **test case value** رو در اختیار نرم افزار قرار دادیم ولی فقط در اختیار نرم افزار می ذاریم **postfix value**ها رو برای اینکه نرم افزار مسیر عادی بعد از تستش رو طی کنه تا به انتهای کارش برسه مثال:

میخوایم چک بکنیم توی سیم کارت که ایا تماس به خارج از کشور برقرار میشه یا نه: توی **prefix**.. مقدار داریم که کلا سیم کارت مکالمات اولیه رو انجام بده تا بررسیم به اون مرحله ای که اماده بشه برای تماس با خارج از کشور خب الان که فهمیدیم ارتباط برقرار شده مثلا دیگه گوشی رو قطع میکنیم یعنی دکمه قطع رو فشار میدیم که این میشه **postfix**.. چون ما میخواستیم وصل شدن رو فقط چک بکنیم و قطع شدنش جز تستمون نبوده و فقط می خواستیم وصل رو چک بکنیم پس موقعی که قطع رو فشار میدیم این جز **postfix**.. هستش و ربطی به تست نداره این ولی برای اینکه نرم افزارمون رو ایجاد خودش رو طی بکنه تا به مرحله خاتمه برسه لازمه که در اختیارش قرار بدیم

نکته: این test case value , postfix value , prefix value در کنار هم یک بخشی از تست کیس میشون و یک بخش دیگش هم Expected results میشه

توی تست کیس هم باید ورودی های مورد انتظار رو داخلش قرار بدیم و هم نتیجه ای که به ازای ورودی ها انتظار داریم رو داخل تست کیس قرار بدیم یعنی Expected results

-----

# Putting Tests Together

## □ Test case

The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software under test

## □ Test set

A set of test cases

## □ Executable test script

A test case that is prepared in a form to be executed automatically on the test software and produce a report

مفاهیم شبهی test set ولی یه تفاوت هایی باهم دارن

test set: یک ستی از تست ها است مثلا تست شماره 1 و شماره 2 ... پس اینا در کنار هم یک مجموعه از تست ها رو تشکیل میده --> پس این یک فایلی است که تو شیه عالمه تست کیس است test sut: اینم یک مجموعه ای از تست است ولی فرقش با بالایی در این است که روی یک فیچر خاصی مرکز شده مثلا همه تست کیس هایی که توی test sut قرار گرفته اند همچون مثلا دارن صفحه لاگین رو تست می کنن ولی test set یک مجموعه ای از تست هاست که مربوط به یک سایکل خاص از نرم افزار است ینی توی مرحله دیزاین این تست کیس ها رو داریم و همچون ممکنه روی یک فیچر مرکز نباشن --> پس می تونیم test sut یک زیر مجموعه ای از test set است

حداقل تعداد تست کیس مورد نیاز برای یک تیکه کد 2 تا است --> یکیش برای پاس شدن ینی رفتارش مثبته و یکیش برای پاس نشدن ینی رفتارش منفیه؟؟ تهش چی شد:/

Executable test script: همون تیکه کدی است که می نویسیم که بهش میگیم که ورودی ها این باشن و خروجی ها هم این باشن

# **Test Automation Framework (3.3)**

A set of assumptions, concepts, and tools  
that support test automation

- فریم ورک:

یک مجموعه ای از راهنمایها + فرضیات مسئله + ابزارهایی که میخوایم استفاده بکنیم که کمک میکنه برای ایجاد یک نرم افزار دیگه = کل این میشه فریم ورک

حالا این فریم ورک کمک میکنه برای ایجاد یک نرم افزار دیگه یا تستش

# What is JUnit?

---

- Open source Java testing framework used to write and run repeatable **automated tests**
- JUnit is open source ([junit.org](http://junit.org))
- A structure for writing **test drivers**
- JUnit **features** include:
  - **Assertions** for testing expected results
  - Test features for sharing **common test data**
  - Test **suites** for easily organizing and running tests
  - Graphical and textual **test runners**
- JUnit is **widely used** in industry
- JUnit can be used as **stand alone** Java programs (**from the command line**) or **within an IDE** such as Eclipse

-

وظیفه اش خودکار سازی تست است **junit** مخصوص زبان جاوا هستش و این **junit** یک فریم ورک است و اپن سورس هست کاری که **junit** انجام میده:

ما می تونیم **junit** یا از کامندلاین ران بکنیم یا این که توی محیط برنامه نویسیمون (توی IDE) به صورت پلاگین ادش بکنیم و رانش بکنیم از اونجا) --> چه خوبیه داره؟ ما می تونیم بهش تست کیس ها رو بدیم و بعد اون خودش میاد **expected result** با خروجی برنامه مقایسه میکنه

Assertions (ادعا): وقتی که کلمه **Assertions** می نویسیم ینی میخوایم ادعا بکنیم این چیزایی که روبروی نوشته شده همواره **True** است

# JUnit Tests

- JUnit can be used to test ...
  - ... an entire object
  - ... part of an object – a method or some interacting methods
  - ... interaction between several objects
- It is primarily intended for unit and integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include :
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests
- Get started at [junit.org](http://junit.org)

برای junit چه مدل تستی استفاده میشه؟

برای unit test : ینی می تونیم کوچکترین واحد برنامه رو توی زبان جاوا باهاش تست بکنیم مثلا متدهای توی یک کلاس یا کل کلاس یا روابط بین کلاس ها

ولی سیستم تست رو نمی تونیم با junit تست بکنیم

وقتی که داریم توی junit تست مینویسیم باید کلاس بنویسیم و بعدی توی کلاس یکسری attribute داریم و یکسری متدهای داریم :

هر متدهای توی junit می نویسیم برای تست کردن یک متدهای توی برنامه هستش ینی این کار درستی نیست که ما با یک متدهای توی junit چندتا متدهای توی برنامه پس برای هر فانکشن ما جدایگانه توی کلاس مربوط به junit میایم متدهای تعریف میکنیم

# Writing Tests for JUnit

- Need to use the methods of the `junit.framework.assert` class
  - javadoc gives a complete description of its capabilities
- Each test method checks a condition (**assertion**) and reports to the test runner whether the test failed or succeeded
- The test runner uses the result to **report to the user** (in command line mode) or update the display (in an IDE)
- All of the methods **return void**
- A few representative methods of `junit.framework.assert`

– `assertTrue (boolean)`

– `assertTrue (String, boolean)`

– `fail (String)`

اینچوری می تونیم **Assertions** رو بنویسیم:

برای دومی: اگر نتیجه مورد انتظار با خروجی تابع یکی بود این بولین میشه `true` در غیر اینصورت میشه `False` و اگر `false` شد اون موقع اون رشته توی کنسول برای تست چاپ میشه مثلایه چیزی بهش میگه

: یک تابعی است که خودش میاد اتوماتیک تست رو به عنوان fail شده اعلام میکنه به تستر ینی خودمون دستی fail می نویسیم و میگیم اعلام کن که fail شده --> این fail توی exception handling بدردمون می خوره --> یکی از روش هایی exception handling این است که اگر توی کد اصلی باید یک exception رو throw بکنه پس باید یک جایی اینو catch بشکنیم و پس اگر بخاطر بررسی exception صدا بزنیم باید منتظر باشیم که یک exception رو بگیره و اگر نگرفت ما خودمون دستی fail رو صدا می زنیم ینی catch نکردیم هیچ چیزی رو و تابع fail صدا زده میشه به عنوان این که اونجا وقتی exception فراهم شد اینجا دیگه صداش نزنیم

نکته: اگر چندتا after , before داشته باشیم هیچ تضمینی برای اینکه با چه ترتیبی اجرا بشن این before , after ها نیست

# JUnit Test Fixtures

---

- A **test fixture** is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (*prefix* values)
  - Reset values (*postfix* values)
- Different tests can **use** the objects without sharing the state
- Objects used in test fixtures should be declared as **instance variables**
- They should be initialized in a **@Before** method
- Can be deallocated or reset in an **@After** method

---

# Simple JUnit Example

```
public class Calc
{
    static public int add (int a, int b)
    {
        return a + b;
    }
}
```

Printed if  
assert fails

Expected  
output

Note: JUnit 4  
syntax

Test  
values

```
import org.junit.Test;
import static org.junit.Assert.*;  
  
public class CalcTest
{
    @Test public void testAdd()
    {
        assertTrue ("Calc sum incorrect",
                    5 == Calc.add (2, 3));
    }
}
```

این boolean اش است

هدف کد: نوشتن یک تابع `add` است که دو تا `int` را میگیره و ریترن میکنه مقدار جمع این دو تا را حالا ما میخوایم برای این تابع `add` بیایم یک تست کیس بنویسیم با استفاده از `junit`:  
ابتدا یک کلاس می نویسیم (توی زبان جاوا همه چیز را داخل کلاس ها می نویسیم)  
راه استاندارد برای نامگذاری کلاس هایی که توی `junit` داریم اینه که اون چیزی که میخوایم تست بکنیم رو اسمش رو می نویسیم و یک تست قبلش یا بعدش اضافه میکنیم مثل `testAdd` یا

## AddTest

توی این کلاس برای این مثال نیازی به `attribute` نداشتیم  
اینجا توی کلاس باید یک متده بنویسیم که متده `add` رو تست بکنه  
وقتی که `@` می ذاریم بهش میگن `annotation` و هدف `annotation` ها اینه که به کامپایلر یا  
محیط ران تایم یکسری توضیح در رابطه با کد می دن و وقتی که `@Test` داریم ینی این تست رو  
به صورت خودکار ران بکن  
همه متدهایی که توی تست می نویسیم مقدار ریترنشون باید `void` باشه

اگر از کلاس `Calc` تابع `add` رو صدابزنم و عدد 2 و 3 رو بهش بدیم این باید با نتیجه مورد انتظار 5 یکی باشه --> اگر مقدار `true` شد که هیچی ولی اگر `false` شد این رشته رو توی صفحه کنسول به تسترنشون میده

وقتی توی یک کلاس تابعی رو به صورت `static` تعریف میکنیم این تابع مربوط به کل کلاس هستش و مربوط به یک `instance` خاص نیست ینی هر وقت که خواستیم صداش بزنیم نباید `instance` بسازیم از کلاس و همینجور کلی می تونیم صداش بزنیم --> اگر داخل همین کلاس که استاتیک فانکشن داره بخوایم صداش بزنیم باید اسمش رو تک و تنها بیاریم مثل `add(2,3)` ولی اگر بخوایم بیرون از کلاس خودش صداش بزنیم اسم کلاس `+` . `+` نام تابع رو میاریم

# Testing the Min Class

```
import java.util.*;  
  
public class Min  
{  
    /**  
     * Returns the minimum element in a list  
     * @param list Comparable list of elements to search  
     * @return the minimum element in the list  
     * @throws NullPointerException if list is null or  
     *         if any list elements are null  
     * @throws ClassCastException if list elements are not mutually  
     *         comparable  
     * @throws IllegalArgumentException if list is empty  
     */  
    ...  
}
```

مثال دوم:

کامنت ها: اینارو اینجوری می نویسیم و بعد می تونیم از ابزارهایی استفاده بکنیم که اتوماتیک داکیومنت این تابع رو تولید میکنه ینی دیگه لازم نیست ما هی تایپ بکنیم توی داکیومنت سازی پس چی کار میکنیم؟ میایم اینارو می نویسیم اینجا و بعد وقتی که از جاوداک استفاده کردیم در قالب یک فایل html داکیومنت مربوط به این کد اماده میشه

هدف اصلی از این کامنت ها اتوماتیک کردن داکیومنت سازی است

یعنی اینترفیس comparable باید پیاده سازی شود توسط سوپرکلاس T

# Testing the Min Class

```
import java.util.List;
public static <T extends Comparable<? super T>> T min (List<? extends T>
list)
{
    if (list.size() == 0)
    {
        throw new IllegalArgumentException ("Min.min");
    }
    Iterator<? extends T> itr = list.iterator();
    T result = itr.next();  
    ایناگی که هایلایت کردم کلمات کلیدی هستن  
    هم یک interface است
    if (result == null) throw new NullPointerException ("Min.min");

    while (itr.hasNext())
    { // throws NPE, CCE as needed
        T comp = itr.next();
        if (comp.compareTo (result) < 0)
        {
            result = comp;  
            اینترفیس comparable است
        }
    }
    return result;  
    یک T از اینترفیس comparable subtype هستش ؟؟
}
```

هدف: از بین عناصر یک لیست مینیم مقدارش رو پیدا بکنیم

اسم تابع `min` است و تایپ خروجیش `generic type` هستش (ینی ما اینو کلی می نویسیم و بعد موقع صدا زدن ینی استفاده کردن از `generic type`ها اون موقع تایپش رو مشخص میکنیم ینی مثلاً می خوایم روی تایپ `int` کار بکنیم)

یک کلمه کلیدی در جاوا است که برای ارث بری استفاده میشه --> میگه وقتی که انتظار داریم تابع `min` تایپش `T` باشه --> `T` ارث بری می کند از کلاسی که `Comparable` رو پیاده سازی می کند

: این یک کلاس هستش که یکی از انواع متدهایی که داره، متدهای داخل اون هدرهاش هست ولی بدنه نداره چرا بدنه خالیه؟ چون میخوایم یک قالب استاندارد داشته باشیم که کلاس هایی که میان اینون اجرا/پیاده سازی میکنن همشون این تابع درونشون مشترک باشه ولی هر کلاسی بسته به ویژگی های خودش پیاده سازی منحصر به فردی داشته باشه مثلاً به ازای وسایل نقلیه مختلف میخوایم یک تابعی داشته باشیم که دور موتور رو حساب میکنه پس یک کلاس `interface` تعریف میکنیم که هدر تابع محاسبه دور موتور هستش و توی `interface` این تابع خالی خالی است بعد توی کلاس موتور سیکلت اینو پیاده سازی می کنیم این `interface` رو و اون موقع می ریم تعریف بدنه این تابع رو میاریم

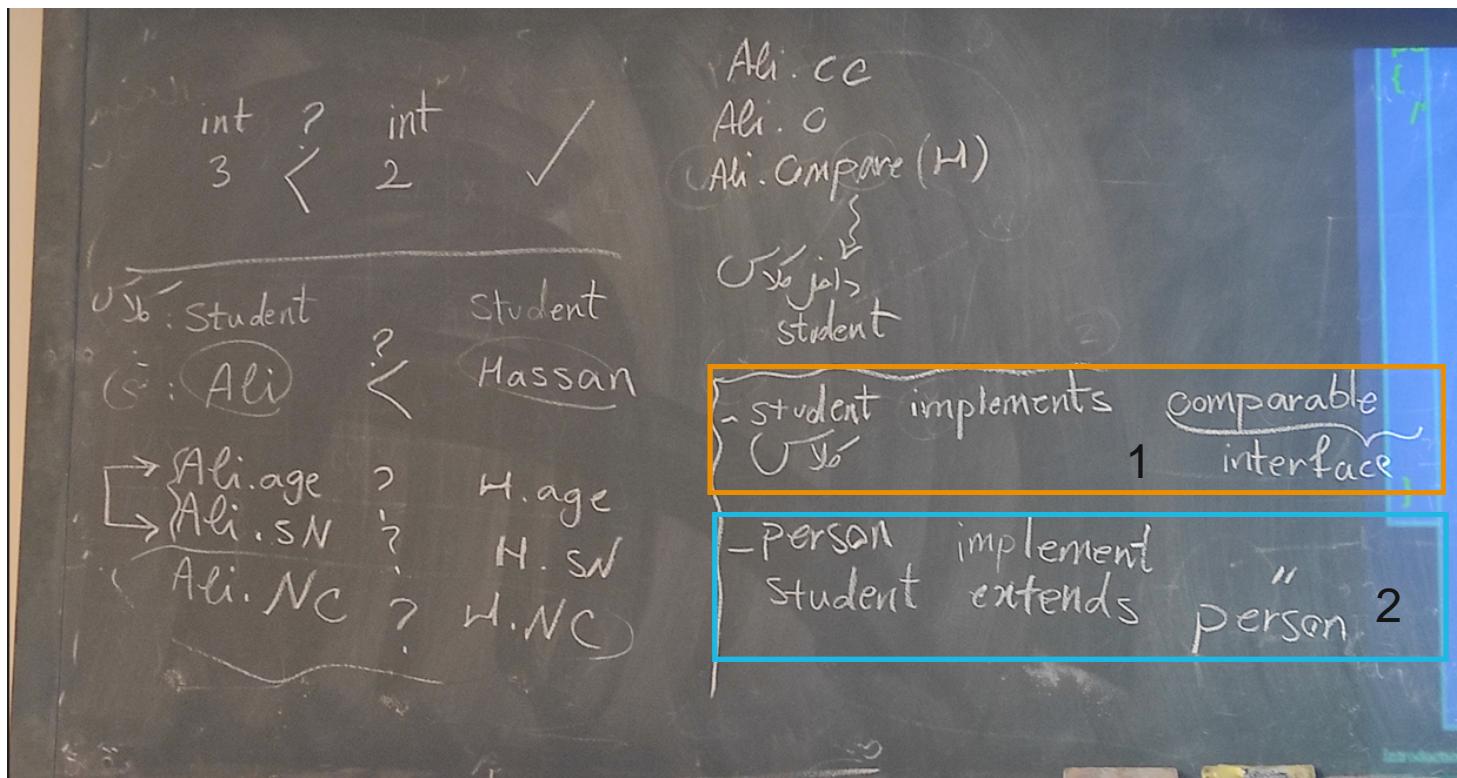
داشتن جز دیزاین پترن هاست که باعث میشه ما یک کد تمیز داشته باشیم کردن ارث بری میکنه `Comparable` رو از سوپر کلاسی که اینترفیس `Comparable` رو داره پیاده سازی میکنه

نکته: یه مدل دیگه متدهایی که توی کلاس اینترفیس داریم اینه که بدنه دارن ولی دیفالت است ینی هر کلاسی که پیاده سازی شد می تونه یا از این همین دیفالت کد استفاده کنه یا اینکه برای خودش جداگانه کد بنویسه

super میگه کدوم کلاس داره اینترفیس Comparable رو پیاده سازی میکنه؟ اون کلاسی که سوپرکلاس T هستش --> در کل همه اینا میگه T یک تایپی است که ابجکت اینها مقایسه شدن با ابجکت هایی از همین نوع رو داره یعنی ابجکت int رو می تونیم با هم نوع های خودش مقایسه بکنیم علامت سوال؟ : یعنی تایپ ناشناخته

ورودی تابع min:

اسمش لیست است و تایپش از جنس لیست است --> ایا لیستی از string است؟ نه چون اگر لیستی از subtype string بود به این صورت میشد: List<string> ولی در اینجا ما می تونیم هر T توی لیست داشته باشیم و subtype همون subclass ها هستن نکته: subtype در زبان جاوا خود کلاس هم شامل میشه یک کلاس است که توابع مختلفی داره Comparable



مثال برای درک بهتر Comparable: عکس در صفحه روبه رو:

مثلا دو تا int داریم یک int داریم 3 و یک int داریم 2 و ایا می تونیم بگیم این رابطه کوچکتری بینشون برقرار هست؟ ایا Comparable هستن؟ بله میشه گفت

حالا یک کلاس student داریم و یک ابجکت ازش میسازیم به اسم علی و برای کلاس بعدی هم حسن ایا می تونیم بگیم علی از حسن کوچکتر است؟ نه تا وقتی که کدش رو ننوشتیم این معنی نداره ینی الان مثلا می خوایم ali.age را بررسی بکنیم در مقایسه با h.age اینو می خوایم مقایسه بکنیم یا علاوه بر این که میخوایم سن ها رو مقایسه بکنیم می خوایم شماره دانشجویی ها هم کوچکتر باشه ینی ایا این هم می خوایم لحاظ بکنیم توی کوچکتر بودن این شی علی از شی حسن؟ --> حالا اینارو کی مشخص میکنه کسی که داره کدنویسی رو انجام میده ینی کسی که داره کدنویسی میکنه میگه هدفش از مقایسه کردن این دو تا ابجکت این هستش که کدمی یکی از دیگری کوچکتر باشه یا نباشه

و ممکنه یکی دیگه هم یک منطق دیگه ای رو پیش ببره توی کدنویسی و... برای این که ما بیایم این کارو انجام بدیم یکی از راه حل ها این هستش که بیایم توی خود کلاس student یک تابع بنویسیم به اسم تابع Compare و توی این تابع بگیم (hassan Compare ali) و اینو بذاریم داخل کلاس Student و

بگیم مثلا وقتی این تابع صدای زده میشه ینی کدمی ها با هم مقایسه بشه (یکی از راه حل هاش اینه)

راه حل بعدی: این برای وقتی که کد بزرگ میشه و استاندارد تر میخوایم باشه از این روش می ریم: استفاده از دیزاین پترن اینترفیس --> کلاس student همون جوری که هست باشه فقط بگو که کلاس:

میکنه ینی اگر خواستیم ابجکت های کلاس Student با هم مقایسه بکنیم از توابعی استفاده میکنیم که پیاده سازی شون داخل کلاس student است ولی اسمشون از کلاس Comparable گرفته شده ینی اسمشون استاندارد است و هر کسی اینطوری نمیتونه برای خودش تابع های دلخواه اسم گذاری کنه

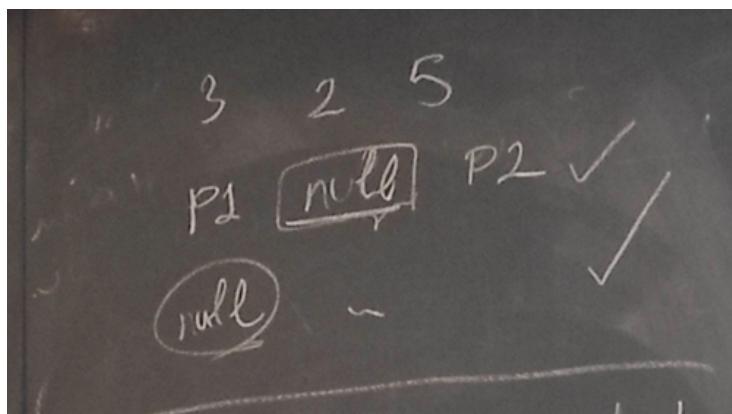
این خودش دو تا راه داره که روی شکل نشون دادم: راه اول رو گفتیم - راه دوم هم میگه می تونه همون حرف بالا باشه ولی student از پدرش بیاد ارث بری کنه و پدرش هم comparable رو پیاده سازی کرده

ادامه کد: توی کد می گه اگه لیست پره که بیاد `min` رو حساب بکنه ولی اگر لیست خالیه `Exception` بده - همینطور اگر داخل لیست هم `null` وجود داشته باشه این قابل قبول نیست و بیا یک `Exception` بده که اینجا برای عنصر اول فقط اینو گفته که اگر عنصر اول `null` بود اینو بگو حالا که مطمئن شدیم لیست خالی نیست باید یک `Iterator` داشته باشیم یعنی یک اشاره گر که روی لیست بچرخه و `min` رو پیدا کنه

هر کدوم از عناصر داخل لیست رو که با `Iterator` نگاه میکنه اون عناصر رو داخل `result` می ذاره پس `result` یک ابجکت از جنس `T` است --> حالا این `result` رو باید با عناصر بعدی لیست مقایسه میکنه ولی یک شرطی داره و اون این است که اگر عنصر اول لیست: `result` اش برابر با `null` بود `exception` بده

حالا مطمئن هستیم که لیست خالی نیست و اولین عنصرش هم `null` نیست پس میایم مرتب اولین عنصری که ریختیم توی `result` با بعدی مقایسه میکنیم

مثالا یک لیستی داریم که 3 و 2 و 5 داخلش هست و به جای `T` موقع صدا زدن `int` می شینه



چیست؟ generic class generic class  
اون هایی هستن که تایپ خروجیشون به جای اینکه  
مثلا بگیم int است میگیم از نوع T است

# MinTest Class

- Standard imports for all JUnit classes :

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

- Test fixture and pre-test setup method (prefix) :

```
private List<String> list; // Test fixture  
  
// Set up - Called before every test method.  
@Before  
public void setUp()  
{  
    list = new ArrayList<String>();  
}
```

- Post test teardown method (postfix) :

اینجا list=null می ذاریم که اماده بشه برای تست بعدی که قراره ران بشه

```
// Tear down - Called after every test method.  
@After  
public void tearDown()  
{  
    list = null; // redundant in this example  
}
```

Before ها قبل از اینکه متد تست اجرا بشه خودشون اتوماتیک اجرا میشن و کارشون این است که به متغیر های fixture مقدار میدن ینی ما یکسری متغیر داریم توی این کلاس تست که بین همه تست ها می خواد مشترک استفاده بشه ولی مقدار دهی اولیشون و اینکه بعد از تست مقدارشون پاک بشه یا اصلاح بشه به عهده تابع های before و after هستش

ما برای تست احتیاج داریم یک لیست تعریف کنیم --> خود این لیست به عنوان اتریبیوت یک کلاس معرفی شده --> هر متد تست میاد قبل از اجرا توسط before لیست رو به رفرنسی از ابجکت ایجاد شده توسط new ArrayList<String> اساین میکنه

@before متدی است قبل از اینکه هر تستی ران بشه میاد به fixture ها مقدار میده   
ArrayList: یک ارایه به طول متغیر هستش که رشته داخلش ریخته شده ینی یک ارایه ای از رشته هاست ولی طول ارایه ثابت نیست

after: بعد از اینکه هر متد تست اجرا شد این چیزایی که بعد از @after هستن میان اتوماتیک اجرا میشن

نکته: توی after , before فقط اون چیزایی که مشترک بین همه است رو قرار میدیم

نکته: اگر چندتا before داشته باشیم همشون اجرا میشن ولی هیچ تضمینی نیست که با چه ترتیبی اجرا می شن

به ازای سه حالت تست را میشیم exception برای ما که مطمئن باشد که اون رو انجام داده exception میشیم

# Min Test Cases: NullPointerException

```
@Test public void testForNullList()
{
    list = null;    کلا لیست مساوی با null باشد
    try {
        Min.min (list); 1
    } catch (NullPointerException e)
        return;
    }
    fail ("NullPointerException expected");
}
```

This NullPointerException test uses the **fail** assertion

This NullPointerException test catches an easily overlooked special case

This NullPointerException test decorates the **@Test** annotation with the class of the exception

```
@Test (expected =
        NullPointerException.class)
public void testForNullElement()
{
    list.add (null); لیست null نیست و دو تا عنصر دارد
    list.add ("cat");
    Min.min (list); ولی اولین عنصرش null است 2
}
```

```
@Test (expected =
        NullPointerException.class)
public void testForSoloNullElement()
{
    list.add (null);
    Min.min (list); لیست null نیست و اولین عنصرش null است و همین یه دونه عنصر رو دارد 2
}
```

توى min ما سه مدل Exception داشتيم --> پس وقتى که تست مى نويسيم باید اینا هم تست بکنیم  
ینی ایا توى شرایطی که باید Exception داده بشه ایا واقعاً Exception داده میشه یا نه  
مهترین بخش تست این هست که حواسمن به Exception ها باشه  
دو روش برای تست کردن جاهايی که Exception داره داده میشه وجود داره:  
روش اول: تابع testForNullList ینی می خوايم یک لیست تهی ایجاد بکنیم --> صدا زدن تابع  
رو توى بلاک try قرار میدیم و گفتیم اگر لیست null بود بیا NullPointerException بگو حالا  
اگر تابع min اشتباه نوشته شده باشه و همچین exception نده ما به عنوان تستر باید اعلام بکنیم  
که اشتباه بود ینی ما منتظر بودیم که exception داده بشه ولی این اتفاق نیوفتد و اسه همین اگر  
توى تابع تست وارد بلاک catch نشدم خودمون میایم دستی fail رو صدا می زنیم و وقتی fail رو  
دستی صدا بزنیم ینی داریم به اون کسی که داشته تست میکرده می گیم ما نتونستیم از طریق  
ریترن کنیم و خودمون دستی او مدیم گفتیم این Exception اصلاً تولید نشد  
روش دوم: وقتی که متده تست رو می نویسیم حالا این متده تستش اینطوری بوده که می خواسته لیستی  
رو تست بکنه که عنصر اول لیست null هستش و عنصر بعدی cat هستش و بعد او مده تابع min  
رو صدا زده روی این لیست

می دونیم قبل از اینکه این متده تست صدا زده بشه before خودش او مده اتوماتیک اجرا شده تابع  
هایی که @before دارن و list مساوی با new ArrayList<String> قرار گرفته و new شده  
و حالا بهش یکی یکی تستر null و cat رو اضافه میکنه به لیست  
روش دوم میشه شماره 2

مثال:

توی مثال زیر اگر  $l=null$  را بذاریم **NullPointerException** داده میشه چون تابع سایز دیگه قابل تعریف روی  $null$  نیست --> اگر  $l=null$  را نداریم لیست هست ولی خالی است در این حالت دیگه **NullPointerException** نداریم

```
List<Integer> l =  
    new ArrayList<Integer>;  
  
if (l.size() == 0) {  
    }  
}
```

برای روش 1:

نکته: وقتی تست پاس میشه که exception واقعا داده بشه پس اگر که throw بشه کار تمومه مهم این است که توی این تست throw نشه در این حالت می گیم کار یه جا خراب است

# More Exception Test Cases for Min

```
@Test (expected =  
ClassCastException.class)  
@SuppressWarnings ("unchecked")  
public void testMutuallyIncomparable()  
{  
    List list = new ArrayList();  
    list.add ("cat");  
    list.add ("dog");  
    list.add (1);  
    Min.min (list);  
}
```

قبل از اینکه هر متد تست اجرا بشه  
قبلش تابع های before اجرا  
میشن--> این یک تابع تست و قبلش  
تابع before اجرا شده و تابع  
before فقط گفته یک لیست بساز  
توی این حالت اگر تابع min روی  
چنین لیستی صدا زده بشه چون  
list.size اش صفر است توی تابع  
exception باید min بده

Note that Java  
generics don't  
prevent clients from  
using raw types!

```
@Test (expected = IllegalArgumentException.class)  
public void testEmptyList()  
{  
    Min.min (list);  
}
```

اینجا سایز لیست صفر است

Special case: Testing for the  
empty list

توی تابع `min` می خوایم حتما عناصر لیست با هم قابل مقایسه باشن --> حالا که با هم قابل مقایسه نیستن رو می خوایم تست بکنیم

اینجا می خوایم یک لیستی رو بسازیم ولی به این صورت ساخته میشه:

`List list = new ArrayList<()-->` وقتی که لیست رو داریم تعریف می کنیم باید برای تایپ عناصرش مشخص بکنیم که چه عنصری رو داخلش داریم می ریزیم مثل اینجا گفته لیستمون با `|` کوچک از تایپ لیستی است که عناصرش داخلش رشته هستن --<`|`

توی زبان جاوا اگر این رشته رو ننویسیم و متغیر رو بدون اینکه تایپ عناصر لیست رو مشخص بکنیم و تعریف بکنیم اتفاقی که می افتد این است که کامپایلر بهمون `Warnings` می دهد ولی این کار مجاز است --> ینی مثلا به جای `|` `List<string>` بگیم `|` و اینجا مشخص نکنیم عناصری که توی لیست قرار میگیرن از چه تایپی هستن این اشتباه است ولی کامپایلر جاوا بهمون خطأ نمیده و فقط `Warnings` می ده و ما `"unchecked@SuppressWarnings` ("unchecked@") می نویسیم ینی به کامپایلر می گیم خودمون حواسمن هست و از عمد اینطوری نوشتم الان لیست رو و تولید نکن

الان که میخوایم یک لیستی بسازیم که هم `int` داره و هم رشته و هم ممکنه چیزای دیگه داخلش باشه موقع تعریف لیست ؟؟

# Remaining Test Cases for Min

```
@Test  
public void testSingleElement()  
{  
    list.add ("cat");  
    Object obj = Min.min (list);  
    assertTrue ("Single Element List", obj.equals  
("cat"));  
}  
  
@Test  
public void testDoubleElement()  
{  
    list.add ("dog");  
    list.add ("cat");  
    Object obj = Min.min (list);  
    assertTrue ("Double Element List", obj.equals  
("cat"));  
}
```

Finally! A couple of  
“Happy Path” tests

---

# **Summary: Seven Tests for Min**

---

- Five tests with exceptions
  - 1. null list
  - 2. null element with multiple elements
  - 3. null single element
  - 4. incomparable types
  - 5. empty elements
- Two without exceptions
  - 6. single element
  - 7. two elements

---

# Data-Driven Tests

---

- Problem : Testing a function multiple times with similar values
  - How to avoid test code bloat?
- Simple example : Adding two numbers
  - Adding a given pair of numbers is just like adding any other pair
  - You really only want to write one test
- Data-driven unit tests call a constructor for each collection of test values
  - Same tests are then run on each set of data values
  - Collection of data values defined by method tagged with @Parameters annotation

---

# Example JUnit Data-Driven Unit Test

```
import org.junit.*;  
import org.junit.runner.RunWith;  
import org.junit.runners.Parameterized;  
import org.junit.runners.Parameterized.Parameters;  
import static org.junit.Assert.*;  
import java.util.*;
```

```
@RunWith (Parameterized.class)  
public class DataDrivenCalcTest  
{ public int a, b, sum;
```

Constructor is called for each triple of values

```
    public DataDrivenCalcTest (int v1, int v2, int expected)  
    { this.a = v1; this.b = v2; this.sum = expected; }
```

مقدارها توی این جدول دو بعدی توی اینجا قرار میگیره

```
    @Parameters public static Collection<Object[]> parameters()  
    { return Arrays.asList (new Object [][] {{1, 1, 2}, {2, 3, 5}}); }
```

Test I  
Test values: 1, 1  
Expected: 2

Test 2  
Test values: 2, 3  
Expected: 5

```
@Test public void additionTest()  
{ assertTrue ("Addition Test", sum == Calc.add (a, b)); }  
}
```

Test method

نکته: قبله به ازای هر تست کیس باید یک متدهست جدگانه می نوشتیم ولی اینجا می تونیم یه عالمه تست کیس داشته باشیم که همشون دارن یک کاری رو انجام میدن مثلاروی اعداد مثبت ۱ با ۱ میشه ۲ و ۲ با ۳ میشه ۵ و ... --> در نهايیت اين کار درست نیست که به ازای هر کدوم از اینا بیایم یک متدهست یا `@Test` بنویسیم چون اینطوری باید صدتا فانکشن بنویسیم و باعث میشه قابلیت نگهداری کد پایین بیاد چون اگر یه زمانی فهمیدیم یه جایی اشتباه شده توی این صدتا کپی پیستی که کردیم باید همه رو تغییر بدیم و اسه همین توی `junit` از `Parameters` استفاده می شه ینی پارامتریک کردن متدهای تست ینی اون `@Test` که قبله می نوشتیم ورودی نمی گرفت متدهای تست ولی الان می تونیم بهشون ورودی بدیم مثال تابع جمع دوتا `:int`

استفاده از `@RunWith` وقتی که داریم `junit` رو پارامتریک می کنیم باید از `Parameters` استفاده کنیم

برای مقداردهی باید یک جدولی (ارایه دو بعدی) ساخته بشه توسط تستر که این جدول یکی میکند مقادیرش خونده میشه توسط `junit` و بررسی میکنه که ایا نتیجه مورد انتظارمون با اون نتیجه ای که میده یکی است یا نه چوری جدول می سازیم با استفاده از یک متدهست `--> قبلاش @Parameters` قرار میدیم که مشخص کنه پارامترهای ما توی اینجا هستن

نکته: ممکنه یک متدهست داشته باشیم که جنس خروجیش متفاوت باشه از ورودی ها در این حالت ارایه رو چوری تعریف میکنیم؟ توی جاوا یک کلاس داریم به اسم `Object` پس اگر یک ارایه دو بعدی از جنس کلاس `Object` تعریف میکنیم در این حالت هر رفرنسی که از ابجکت با جنس های مختلف توی ارایه دو بعدی باشه قابل قبوله پس اگر بنویسیم `Object` ینی تمام تایپ های ممکن قابل قبوله

ینی `typecast` میکنیم ارایه رو به لیست `Arrays.asList`

یک utility class `Arrays` است توی جاوا--> utility class ها یک سری فانکشن هستن که ما میتوانیم استفاده بکنیم از شون و فانکشن هاش هم به صورت استاتیک تعریف شدن مثل از این utility class می تونیم استفاده بکنیم که یکسری اعمال رو ارایه ورودی انجام بدیم و یکسری اپریشن دارن مثل می تونیم بگیم `Arrays.sort` یا ... پس فقط اپریشن داره و هیچ attribute داخلش نیست و اپریشن هاش هم به صورت استاتیک است

هم میاد ارایه دو بعدی که ساختیم از جنس ابجکت ها رو به لیست تبدیلش میکنه `aslist` یک کلاس اینترفیس است مثل `set` و `list` و... که خودشون یک زیراینترفیس کلاس `Collection` هستن `Collection` سوال:

چرا خروجی `Collection` گفته ولی ما چیزی که ریترن میکنیم با استفاده از این utility class همین لیست است؟ چون خود لیست یک `Collection` است

وقتی `junit` رو ران میکنیم مثل قبل میاد هر تابع `@Test` رو اتوماتیک ران میکنه ولی چون با استفاده از `RunWith(Parameterized.class)` داریم رانش میکنیم قبلش میاد نگاه میکنه با اونایی که توی جدولی که قبلا ساختیم و به ازای هر سطر توی جدول میاد اتوماتیک مقادیر توی جدول رو انتساب میده به متغیرهای `constructor` کلاسمون پس اینجا یک هم یک سازنده نیاز داریم

پس در حالت کلی موقع پارامتریک کردن به سه چیز نیاز داریم: 1- تابع `@Test` با پارامتر بنویسیم 2- جدول ورودی ها رو مشخص بکنیم 3- سازنده کلاس رو بنویسیم سازنده کلاس میاد به ترتیب هر سطر رو که توی جدول داریم می گیره و داخل متغیرهای خودش می ریزه

الان اینجا سازنده کلاس `DataDrivenCalcTest` است ؟؟

نکته: وقتی `junit` رو ران بکنیم خودش میاد به ازای هر سطر توی جدول، سازنده رو صدا می زنه

خلاصه:

@Parameters: مقدار این پارامترها رو توی یک جدول مشخص میکنیم حالا یک جدول دو بعدی می سازیم با استفاده از یک تابع که این تابع رو با استفاده از @Parameters مشخص میکنیم و توی هر سطر از ای ارایه دو بعدی مقدار `a, b, sum` مشخص میشه و وقتی که `junit` رو ران میکنیم به ازای هر سطرها سازنده ران میشه و همینطور تابع @Test هم ران میشه پشتیش به ازای هر سطر

نکته: تابع @Parameters داره قبل از سازنده ران میشه پس یک تابع وقتی که بخواهد قبل از سازنده ران باشه، هنوز هیچ instance ازش ساخته نشده پس باید استاتیک تعریف بشه وقتی یک instance از کلاس ساخته میشه که سازنده در حال اجرا باشه پس اگر قبل از سازنده یک تابعی رو ران بکنیم باید استاتیک تعریفش بکنیم که نخواهد وابسته به وجود instance باشه

نکته: وقتی که اسم کلاس + دات بذاریم ینی لازم نیست instance بسازیم و متده استاتیک بوده

# Tests with Parameters: JUnit Theories

- Unit tests can have actual parameters
  - So far, we've only seen parameterless test methods
- Contract model: Assume, Act, Assert
  - *Assumptions* (preconditions) limit values appropriately
  - *Action* performs activity under scrutiny
  - *Assertions* (postconditions) check result

```
@Theory public void removeThenAddDoesNotChangeSet ( //  
    Set<String> someSet, String str) {  
  
    Parameters!  
    assumeTrue (someSet != null) // Assume  
    assumeTrue (someSet.contains (str)) ; // Assume  
    Set<String> copy = new HashSet<String>(someSet); // Act  
    copy.remove (str);  
    copy.add (str);  
    assertTrue (someSet.equals (copy)); // Assert
```

@Theory: ینی قانون های کلی که توی برنامه جاری است رو می خوایم چک بکنیم مثلاً ما می دونیم یک قانونی توی برنامه است مثلاً اومدیم توابع مربوط به رشته ها رو پیاده سازی کردیم و گفتیم اگر این تابع هایی که کلا رشته ها رو پردازش میکنند درست نوشته باشیم مثلاً الان اینجا add و remove داریم --> اگر یک زیررشته ای رو از توی یک رشته حذف کنیم و بعد دوباره همون جا اضافه اش کنیم نتیجه ابتدایی با نتیجه انتها یی باید یکی باشه

توی @Theory ها یک پیش شرط داریم و وقتی که روی پیش شرط یک اکشنی انجام میشه باید حتماً postconditions رو ببینیم و اگر نبینیم تستمون fail میشه --> اینا با استفاده از سه بخش Assume, Act, Assert مطرح میشه:

چجوری می نویسیم؟ میایم یک تابع می نویسیم توی همون قسمت تست ها به اسم @Theory الان میخواهد چک بکنه اگر یک ستی داشته باشه ینی یک مجموعه ای از رشته های مختلف داشته باشه و بیاد از توی این مجموعه از رشته های یه دونه رشته رو حذف کنه و بعد دوباره همونو اضافه کنه نتیجه انتها یی با ست اولیه پکسان باشه

حالا این تابع به چه preconditions یا پیش شرط هایی احتیاج داره: میخوایم اگر یکی از ورودی ها عضو این ست بود بعد چک بکنیم با اکشن های حذف و اضافه کردن نتیجه انتها یی با اون ست اولیه متفاوت هست یا نیست

اول از همه `assumeTrue` را مشخص میکنیم با استفاده از تابع `assumeTrue` و فرض میکنیم درست هستش

حالا این تابع `assumeTrue` دو تا ورودی دارد: یکی `str` است و یکی هم `someSet` و `someSet` یک سنتی هستش از رشته ها

یک اینترفیس است در زبان جاوا --> که این `set` زیر اینترفیس کلاس `Collection` است و عضو تکراری نداره

حالا اینجا فرض کرده سمت مجموعه ای از رشته ها است و اسمش رو گذاشته `someSet` پس فرض اولمون اینه که این رشته یا `str` توی این مجموعه `someSet` وجود داشته باشه : این فرض رو گرفتیم درست

یک متده است که توی اینترفیس `set` وجود داشته

توی زبان جاوا داره `implement` میکنه اینترفیس `set`

بعد اگر `str` عضو سمت بود میاد یکی از سمت میسازه بعد از توی اون کپی رشته مورد نظر را حذف کرده و بعد رشته رو اضافه کرده و الان میخواد چک بکنه ایا اون رشته ای که این تغییراتو روش اعمال کردیم با رشته اصلی که `someSet` است یکی بوده یا نه

اگر `HashSet` یک کلاس است و `implement` کرده ینی او مده توابعی که توی اینترفیس `set` بوده رو بدنشون رو نوشته

اگر به جای `HashSet` می او مدیم `set` رو می نوشتیم چی میشد؟ توی `set` ما بدنه نداریم ینی اگر روی کپی تابع حذف رو صدا می زدیم این تابع بدنه نداشت ولی توی `HashSet` تابع ها بدنه دارن

توضیح بهتر راجع به :**HashSet**

توی کلاس **HashSet** یکسری تابع داخلش هست مثل تابع حذف و اضافه کردن

این تابع ها از کجا او مده؟ اینا توی اینترفیس **set** بوده و ما موقع تعریف کلاس **HashSet** گفتیم این کلاس **HashSet** داره **implement** میکنه **set** پس می دونیم تابع حذف و اضافه کردن رو **set** داشته ولی بدنه رو نداشته و او مدیم بدنه رو توی کلاس **HashSet** نوشتیم

# Question: Where Do The Data

## □ Answer: Values Come From?

- All combinations of values from @DataPoints annotations where assume clause is true
- Four (of nine) combinations in this particular case
- Note: @DataPoints format is an array

```
@DataPoints
```

```
public static String[] animals = {"ant", "bat", "cat"};
```

```
@DataPoints
```

```
public static Set[] animalSets = {  
    new HashSet((Arrays.asList("ant", "bat")),  
    new HashSet((Arrays.asList("bat", "cat", "dog", "elk")),  
    new HashSet((Arrays.asList("Snap", "Crackle", "Pop")))  
};
```

Nine combinations of  
animalSets[i].contains (animals[j])  
is false for five combinations

برای تابع هایی که @Theory دارن میایم @DataPoints می نویسیم  
@DataPoints متدهایی نیست --> این @DataPoints یه دونه ارایه می سازه --> به ازای هر ورودی باید یکبار @DataPoints رو بنویسیم و به ازای اون زیر عضو set مون هم که یک رشته بود باید یک ارایه دیگه بسازیم --> پس یک ارایه از رشته ها و یک ارایه از set ها میسازیم

تابع @Theory دوتا ورودی داره Set<String> someSet, String str حالا به ازای هر کدام از این ورودی ها جداگانه @DataPoints رو می نویسیم و هر کدام از @DataPoints برای یک کدام از این ورودی ها هستن --> و برای هر کدام از اینا که یک @DataPoints می نویسیم به صورت یک ارایه می نویسیم

حالا چند بار تابع ران میشه به ازای این DataPoints ها ؟ 9 بار --> چون سه تا رشته مختلف داریم و سه تا سمت مختلف داریم پس  $3^3 = 27$  میشه

# JUnit Theories Need BoilerPlate

```
import org.junit.*;  
import org.junit.runner.RunWith;  
import static org.junit.Assert.*;  
import static org.junit.Assume.*;  
  
import org.junit.experimental.theories.DataPoint;  
import org.junit.experimental.theories.DataPoints;  
import org.junit.experimental.theories.Theories;  
import org.junit.experimental.theories.Theory;  
  
import java.util.*;  
  
@RunWith(Theories.class)  
public class SetTheoryTest  
{  
    ... // See Earlier Slides  
}
```

---

# Running from a Command Line

- This is all we need to run JUnit in an IDE (like Eclipse)
- We need a `main()` for command line execution ...

---

# AllTests

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import junit.framework.JUnit4TestAdapter;

// This section declares all of the test classes in the program.
@RunWith (Suite.class)
@Suite.SuiteClasses ({ StackTest.class }) // Add test classes here.

public class AllTests
{
    // Execution begins in main(). This test class executes a
    // test runner that tells the tester if any fail.
    public static void main (String[] args)
    {
        junit.textui.TestRunner.run (suite());
    }

    // The suite() method helps when using JUnit 3 Test Runners or Ant.
    public static junit.framework.Test suite()
    {
        return new JUnit4TestAdapter (AllTests.class);
    }
}
```

---

# JUnit 5 changes: min() Example

- JUnit 5 uses assertions, not annotations, for exceptions

```
@Test public void testForNullList()
{
    assertThrows(NullPointerException.class, () -> Min.min(null));
}
```

- Other JUnit 5 differences
  - Java lambda expressions play a role
  - @Before, @After change to @BeforeEach, @AfterEach
  - imports, some assertions change
  - Test runners change (no simple replacement for AllTests.java)
  - @Theory construct moved to 3<sup>rd</sup> party extensions
    - google “property based testing”
- See MinTestJUnit5.java on the book website

---

# How to Run Tests

---

- JUnit provides **test drivers**
  - **Character-based** test driver runs from the command line
  - **GUI-based** test driver-*junit.swingui.TestRunner*
    - Allows programmer to specify the test class to run
    - Creates a “**Run**” button
- If a test fails, JUnit gives the **location** of the failure and any **exceptions** that were thrown

---

# JUnit Resources

---

## □ Some JUnit tutorials

- <http://open.ncsu.edu/se/tutorials/junit/>  
(Laurie Williams, Dright Ho, and Sarah Smith )
- <http://www.laliluna.de/eclipse-junit-testing-tutorial.html>  
(Sascha Wolski and Sebastian Hennebrueder)
- <http://www.diasparsoftware.com/template.php?content=jUnitStarterGuide>  
(Diaspar software)
- <http://www.clarkware.com/articles/JUnitPrimer.html>  
(Clarkware consulting)

## □ JUnit: Download, Documentation

- <http://www.junit.org/>

---

# Summary

---

- The only way to make testing **efficient** as well as **effective** is to **automate** as much as possible
- Test frameworks provide very simple ways to **automate** our tests
- It is no “**silver bullet**” however ... it does not solve the hard problem of testing :

**What test values to use ?**

- This is test design ... the purpose of test criteria

---